UOST: UML/OCL Aggressive Slicing Technique for Efficient Verification of Models

Asadullah Shaikh, Uffe Kock Wiil, and Nasrullah Memon

The Maersk Mc-Kinney Moller Institute
University of Southern Denmark
{ashaikh,ukwiil,memon}@mmmi.sdu.dk

Abstract. In Model Driven Development (MDD), model errors are a primary concern in development methodology. UML/OCL models have increased both in size and complexity due to its popularity in model design and development. Consequently, the efficiency of the verification process is being affected. The verification of these models is each time more laborious because of their complex design and size thus prolonging the verification process. In this paper, we propose an algorithm of an aggressive slicing technique that works on UML/OCL models (a collection of classes, associations, inheritance hierarchies, and OCL constraints) which improves the efficiency of the verification process. With this technique, the submodels of an original model are computed through partitioning those classes and multiplicities that are not used in written constraints and at the same time, ensuring that the model behavior is not affected. We attempt to quantify the speed-up achieved by adding the slicing technique to two different tools (1) UMLtoCSP and (2) Alloy. The purpose behind showing the results in UMLToCSP and Alloy is to demonstrate that the developed slicing technique is neither tool dependent nor formalism dependent.

Keywords: Non Disjoint UML/OCL Slicing, Model Slicing, Verification of UML/OCL Models, Non Disjoint Slicing of UML/OCL Class Diagrams.

1 Introduction

UML/OCL models are designed in order to provide a high-level description of a software system which can be used as a piece of documentation or as an intermediate step in the software development process. In the context of Model Driven Development (MDD) and Model Driven Architecture (MDA), a correct specification is required because all the technology is based on model transformation. Therefore, if the original model is wrong, this clearly causes a failure of the final software system. Regrettably, verification of a software product is a complex and time consuming task [6] and that also applies to the analysis of the software models. With increasing model size and complexity, the need for efficient verification methods able to cope with the growing difficulties is ever present, and the importance of UML models has increased significantly [2].

F.A. Kraemer and P. Herrmann (Eds.): SAM 2010, LNCS 6598, pp. 173–192, 2011.
© Springer-Verlag Berlin Heidelberg 2011

At present, we are facing efficiency problems when verifying OCL constraints of complex 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. One possible approach is *slicing*; which is partitioning the class diagram and OCL constraints into smaller fragments according to certain criteria. This partition should preserve the property under verification in the sense that it should be possible to assess the property in the original model from the analysis of the partitions. A careful definition of the partition process is needed to ensure this.

We focus our discussion on the verification of a specific property: satisfiability, i.e., "is it possible to create objects without violating any constraint?" The property is relevant in the sense that many interesting properties, e.g., redundancy of an integrity constraint, can be expressed in terms of satisfiability [5]. Two different notions of satisfiability can be checked: either weak satisfiability or strong satisfiability. A class diagram is weakly satisfiable if it is possible to create a legal instance/object of a class diagram which is non-empty, i.e., it contains at least one object from some class. Alternatively, strong satisfiability is a more restrictive condition requiring that the legal instance has at least one object from each class and a link from each association [4]. A few slicing techniques exist that break large models into smaller segments, however, those techniques do not address the verifiability of UML/OCL models. Previous research on slicing has focused on slicing methods of UML architectural models and verifying invariants in pieces [1, 13, 10, 15, 11].

In this paper, we propose an aggressive slicing technique which preserves the satisfiability of the model after partitioning. The technique improves the efficiency of the verification process for large and complex UML/OCL models. The technique includes a set of heuristics that are used to partition a model when determining its satisfiability. That is, given a model 'm', the technique partitions m into m_1, m_2, m_3,...m_n submodels, where m is satisfiable if all m_1, m_2, m_3,...m_n submodels are satisfiable. This slicing technique is called the UML/OCL Slicing Technique (UOST). We provide an experimental evaluation of this technique using a verification tool for UML to CSP which is called UMLtoCSP [4] and Alloy [8]. We examine both small and large UML/OCL models with 2, 15, 50, 100, 500, and 1000 UML/OCL class diagrams and several OCL invariants to measure the efficiency of the verification process through our proposed UOST. The original explanation of the slicing technique can be found in [17]. However, this paper presents a solution for a non disjoint set of submodels using a more aggressive slicing technique that can still preserve the satisfiability of the model after partitioning. It also provides extensive results achieved by adding the slicing technique to an external tool (Alloy). The primary reason for showing the results in both Alloy and UMLtoCSP is to demonstrate that the developed slicing technique is neither tool dependent nor formalism dependent. It can be applied into any formal verification tool for UML/OCL models.

The rest of the paper is structured as follows. Section 2 describes the concepts of UML/OCL model slicing. Section 3 presents the solution for non disjoint submodels. Sections 4 and 5 present the results obtained from model slicing. Section 6 reviews some previous work related to slicing. Finally, Sect. 7 presents conclusions and future work.

2 UML/OCL Model Slicing

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

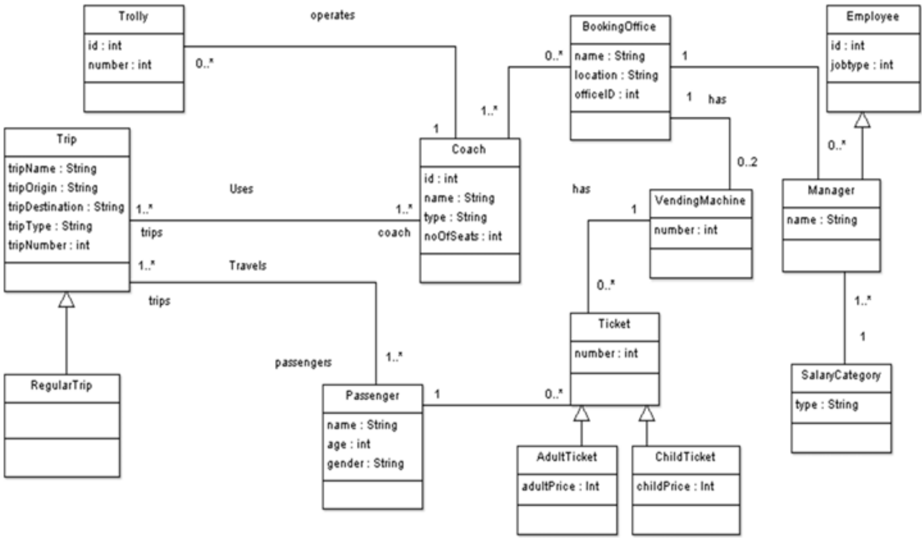
Two different notions of satisfiability will be considered for verification: *strong* satisfiability and *weak* satisfiability. A class diagram is weakly satisfiable if it is possible to create at least one instance of at least one class out of all classes in the class diagram. Alternatively, in the case of strong satisfiability, it is an obligation that at least one object of all classes must be instantiated [4]. For example, it is possible that objects of all classes are not instantiated due to multiple inheritance, composition, and aggregation. In this case, the model will be considered as unsatisfiable in the case of strong satisfiability. Consequently, strong satisfiability requires the existence of an object for each concrete subclass of an abstract class.

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

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

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

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



context Coach **inv** passengerSize : self.trips−>select(r|r.oclIsTypeOf(RegularTrip))−>forAll(t|t.passengers −>size() $\leq$ noOfSeats)
context Ticket **inv** ticketNumberPositive: self.number $>$ 0
context Passenger **inv** NonNegativeAge: self.age $\geq$ 0

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

2.1 The UOST Process

The method introduced for computing UML/OCL slicing is shown in Fig. 2. The process begins in step 1 by identifying the classes, associations, and generalizations created in model M and subsequently, drawing a flowgraph. In step 2, we identify OCL invariants and group them if they restrict the same model elements. We call this "clustering of constraints" (Constraint Support). The constraint support defines the scope of a constraint. The support information can be used to partition a set of OCL invariants into a set of independent *clusters* of constraints, where each cluster can be verified separately. The following procedure is used to compute the clusters:

- Compute the constraint support of each invariant.
- Keep each constraint in a different cluster.
- Select two constraints x and y with non disjoint constraint supports and located in different clusters, and merge those clusters.
- Repeat the previous step until all pairs of constraints with non disjoint constraint supports belong to the same cluster.

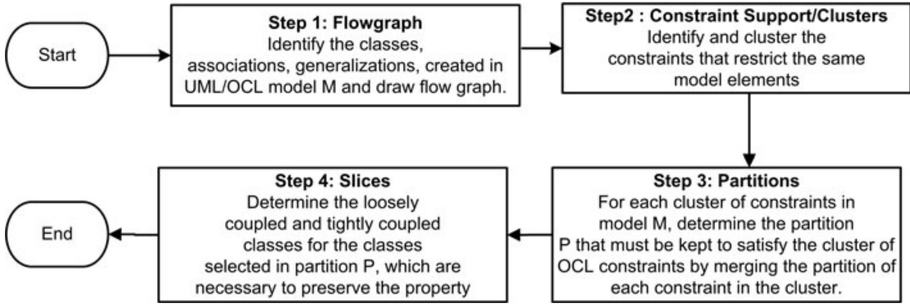


Fig. 2. UOST Process Steps

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

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

2.2 Flowgraph: Step 1

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

By applying step 1 (Fig. 2), we build a flowgraph based on the identification of classes, associations, and generalizations as shown in Fig. 3. We use the concept of a flowgraph to capture the dependencies among model elements. This concept is also used by other slicing approaches [18,14,16]. A flowgraph is a set of vertices and directed arcs where the vertices represent classes from a class diagram and the arcs model relationships between these classes. In our approach, a flowgraph contains vertices and arcs for each pair of classes connected by associations, generalizations, aggregations, or compositions.

We consider two types of relationships among classes; tightly associated and loosely associated classes. These relationships attempt to capture the necessity of creating instances of one class when an instance of the other exists. Loosely coupled classes have an association with a lower bound of 0 (e.g., 0..3); this

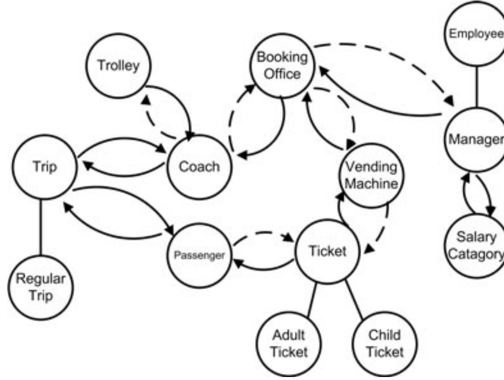


Fig. 3. Flowgraph of model Coach

Table 1. Loosely and tightly coupled classes

UML relationship	*Loosely/Tightly Coupled*	*Arc/Edge*
Association: Lower bound ≥ 1 (e.g., 1..*)	Tightly Coupled	⟶
Association: Lower bound = 0 (e.g., 0..3)	Loosely Coupled	- - →
Generalization, Aggregation, and Composition	Tightly Coupled	——

means if an object of class A is instantiated, then it is not necessary that an object of class B must be instantiated. Tightly coupled classes are the inverse of loosely coupled classes, i.e., they have an association with a lower bound greater than 1 (e.g., 1..*).

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

2.3 Applying UOST: Step 2, Step 3, and Step 4

In this section, we compute constraint support, partitions and form the final slices for verifiability.

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

Algorithm 1. Flowgraph creation

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

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

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

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

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

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

Algorithm 2. Slicing Algorithm

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

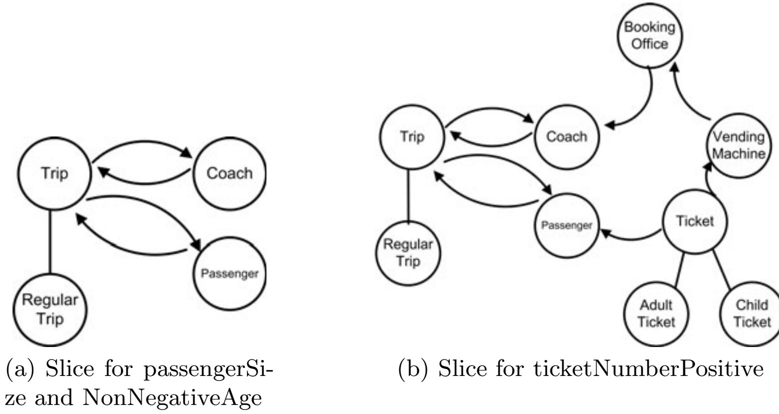
```
1: G ← BuildFlowGraph(M) {Creating the flowgraph}
2: {Cluster the OCL constraints}
3: for each pair of constraints c1, c2 in M do
4:   if ConstraintSupport(M, c1) ∩ ConstraintSupport(M, c2) ≠ ∅ then
5:     MergeInSameCluster(c1, c2)
6:   end if
7: end for
8: {Work on each cluster of constraints separately}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

- The classes and relationships in the cluster of constraint supports are part of the slice.
- Any class with a tightly coupled relationship to a class in the slice is also a part of the slice, as is the relationship.

3 Non Disjoint Solution

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

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



(a) Slice for passengerSize and NonNegativeAge

(b) Slice for ticketNumberPositive

Fig. 4. Slices of s1 and s2

- $C_1 \in C$ is a class.
- $C_2 \in C$ is a class.
- m_1 and m_2 are non-negative integers $\in Z^+$ where m_1 and m_2 correspond to the lower bound of the multiplicity of each association end for C_1 and C_2, respectively.
- M_1 and M_2 are non-negative integers or infinity ($M_1 \in (Z^+ \bigcup \{\infty\})$) where M_1 and M_2 corresponds to the upper bound of the multiplicity of each association end for C_1 and C_2, respectively, and $M_i \geq m_i$.

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

- $C' \in C$
- $R' \in R$
- $A' \in A$
- $c \in C' \rightarrow A_c \subseteq A'$
- $(C_1, C_2, m_1, M_1, m_2, M_2) \in R' \rightarrow C_1, C_2 \in C'$

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

- Satisfiability (Strong/Weak): If the objects of a given class C in a submodel S are instantiated as per given expression in the cluster of OCL constraints CL_c, then submodel S is satisfiable.
- Unsatisfiability: If there are two or more constraints whose interaction is unsatisfiable, then submodel S is also unsatisfiable. It indicates that some expression in the OCL invariant is violated and that the objects of the classes cannot be instantiated according to the given OCL expression.

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

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

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

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

We present the non disjoint solution if slicing is applied over a UML model without clustering the constraints (i.e., without step 2 in the UOST process). There are three major steps that need to be considered as a solution:

- Find the common class in all slices of the Model (M).
- For each constraint, find the maximum of the lower bound (m_1) multiplicities relevant to the constraint from all associations of the common class. Set this maximum as the base value. $Base_c = max(m_1)$ where $(c, C_2, m_1, M_1, m_2, M_2) \in R$.
- Compare the base value using the expression given in each constraint.

context Researcher **inv** NoSelfReviews: :
self.submission −>excludes(self.manuscript)

context Paper **inv** AuthorsOfStudentPaper:
self.studentPaper = self.author −>exists(x | x.isStudent)

context Paper **inv** NoStudentReviewers:
self.referee −>forAll(r | not r.isStudent)

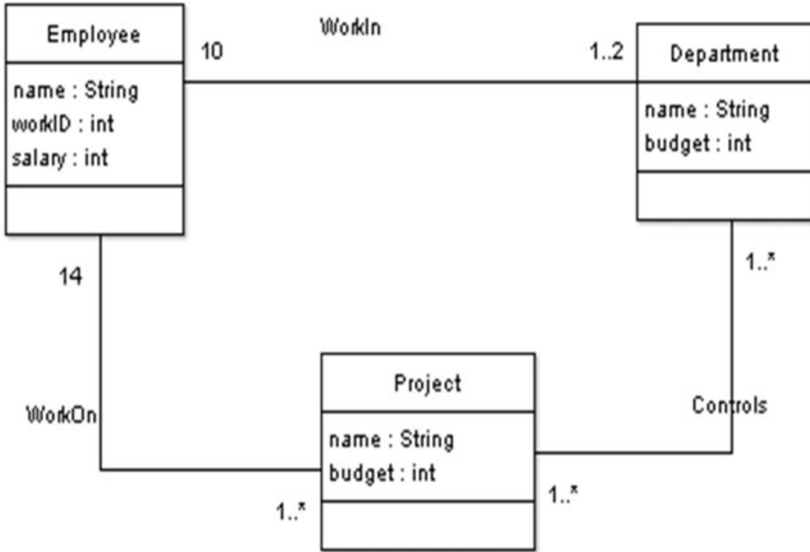
context Paper **inv** LimitsOnStudentPapers:
Paper::allInstances()−>exists(p | p.studentPaper) and
Paper::allInstances()−>select(p | p.studentPaper) −>size() < 5

Fig. 5. UML/OCL class diagram of 'Paper-Researcher' [3]

The OCL constraints can be either textual OCL invariants or graphical restrictions like multiplicities of association ends. This property is important not only because it can point out inconsistent models, but also because it can be used to check other interesting properties like the redundancy of an integrity constraint. For example, there could be a case where the invariants are constrained from the same class of the model. Figure 6 introduces a class diagram of 'model Company' used to exemplify our non disjoint solution. There are two constraints *departmentEmployeeSize* and *projectEmployeeSize* whose properties need to be checked. Invariant *departmentEmployeeSize* is satisfiable however, invariant *projectEmployeeSize* is unsatisfiable due to a violation of multiplicity. After applying the slicing technique without clustering the invariants, we will receive two submodels i.e., two non disjoint slices. Slice 1 will consist of class 'Department' and class 'Employee' for constraint *departmentEmployeeSize*. Similarly, class 'Project' and class 'Employee' for invariant *projectEmployeeSize* will be part of slice 2.

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

Initially, our non disjoint approach finds the common class in all slices of model (M), i.e., class 'Employee'. Secondly, the method finds the maximum of minimum (max_min) multiplicities from the common class (Employee) for each constraint considering its navigation. For example, the navigation of invariant "departmentEmployeeSize" is class 'Department' navigating to class 'Employee'. Therefore, the approach considers the multiplicity between the navigation of



context Department **inv** departmentEmployeeSize :
self.employee()−>size() = 10

context Project **inv** projectEmployeeSize:
self.employee()−>size() ≥ 15

Fig. 6. UML/OCL class diagram used as non disjoint solution (model Company)

class department and class employee, i.e., ‘10’ and ‘1..2’. As the constraint restricts class employee, ‘10’ is the base value for the “departmentEmployeeSize” invariant. Similarly, ‘14’ is the base value for the navigation of class ‘Project’ and class ‘Employee’.

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

4 UOST Implementation in UMLtoCSP

We have implemented our proposed slicing technique (UOST) in UMLtoCSP [4] in order to show the improvement of the efficiency in the verification process. After developing UOST, we named our tool as UMLtoCSP(UOST). The execution time of verification of an original UMLtoCSP depends mainly on the number of classes/attributes and the parameters offered during the transformation to the Constraint Satisfaction Problem (CSP). In case of small models, UMLtoCSP

Table 2. Description of the examples

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

Table 3. Description of experimental results (UMLtoCSP)

Before Slicing (UMLtoCSP)			After Slicing (UMLtoCSP UOST)				
Classess	Attributes	OVT	Attributes	ST	SVT	TVT	Speedup %
2	6	2506.55s	3	0.00s	0.421s	0.421s	99.98%
15	2	5008.76s	0	0.00s	0.178s	0.178s	99.99%
50	72	3605.35s	55	0.016s	0.031s	0.047s	99.99%
100	122	Time out	117	0.016s	0.032s	0.048s	99.99%
500	522	Time out	502	0.062s	0.028s	0.090s	99.99%
1000	1022	Time out	1012	0.282s	0.339s	0.621s	99.98%

OVT Original Verification Time **ST** Slicing Time
SVT Sliced Verification Time **TVT** Total Verification Time

provides quick results while for larger ones, the tool takes a huge of amount of time. In order to evaluate the efficiency of our developed UOST approach in UMLtoCSP, several models have been used (Table 2). UMLtoCSP takes a lot of time to verify the instances of large examples, therefore, we set a time out to 1 hour 30 minutes which is equal to 5400 seconds. If UMLtoCSP does not verify the model in the prescribed time, we will count this situation as *Time out*.

Table 3 summarizes the experimental results obtained by UMLtoCSP and UMLtoCSP(UOST) running on an Intel Core 2 Duo 2.10 Ghz with 2Gb of RAM where, the column OVT is the original verification time of UMLtoCSP, column TVT is the total verification time of all slices of UMLtoCSP(UOST), and column speedup shows the efficiency obtained after the implementation of the slicing approach. We have used the following parameters for the experiments: each class may have at most 4 instances, associations may have at most 1010 links and attributes may range from 0 to 1022. The speedup is calculated using the equation $\{1 - (TVT/OVT)\} * 100$.

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

4.1 Limitations

Our proposed technique is limited and cannot partition the UML/OCL model and abstract the attributes if the constraints are restricted from all classes using all attributes of the class diagram. In this case, the technique will consider a

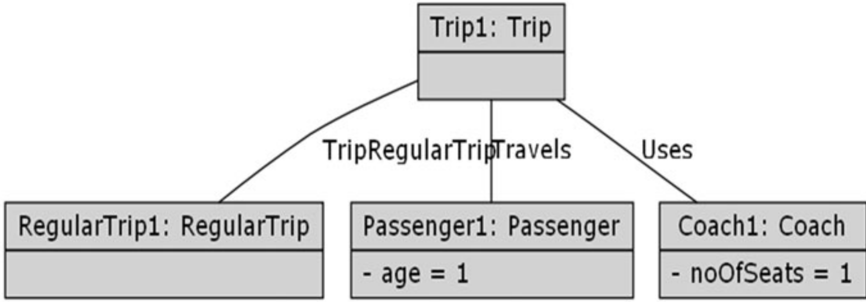


Fig. 7. Submodel 1 (s1) object diagram for strong satisfiability

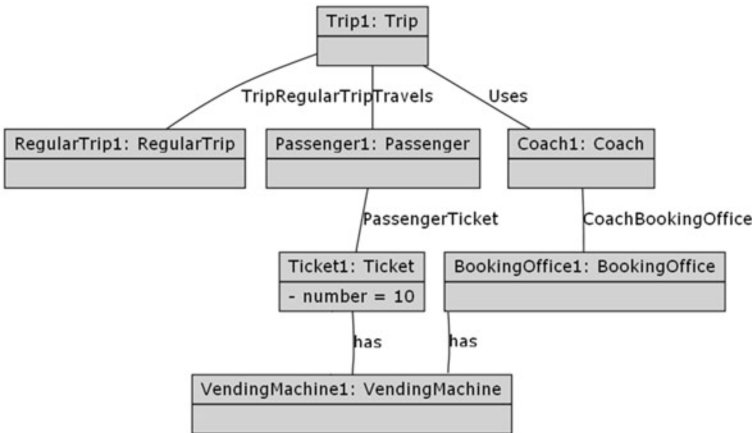


Fig. 8. Submodel 2 (s2) for weak satisfiability

UML/OCL model as a single model and therefore, there will be no difference between UMLtoCSP and UMLtoCSP(UOST). Table 4 describes the worst case examples that cannot be sliced using UOST. The example *Paper-Researcher* is a real world-example created manually which contains 2 classes, 6 attributes, 2 associations, and 4 invariants while example *Company* is script-generated and has 100 classes, 100 attributes, 100 associations, and 100 invariants. In these examples, partitioning cannot be done by the proposed UOST technique because each instance of a class is restricted by an invariant.

5 UOST Implementation in Alloy

In this section, we present several examples in the Alloy specification in order to prove that our developed slicing technique is neither tool dependent nor formalism dependent. We compare the verification time of several UML/OCL class

Table 4. Worst case examples

Tool & Example	Classes	Associations	Attr	Inv	NOS	OVT
UMLtoCSP (Paper-Researcher)	2	2	5	4	0	0.040 s
UMLtoCSP(UOST) (Paper-Researcher)	2	2	5	4	0	0.036 s
UMLtoCSP (Company)	100	100	100	100	0	0.070 s
UMLtoCSP(UOST) (Company)	100	100	100	100	0	0.078 s

Attr Associatons **Inv** Invariants
NOS Number of Slices **OVT** Original Verification Time

Table 5. Description of the examples

Example	Classes	Associations	Attributes	Invariants
Atom-Molecule	2	2	3	2
University	4	3	8	5
ATM Machine	50	51	51	7
Script 1	100	110	122	2
Script 2	500	510	522	5
Script 3	1000	1010	1022	5

diagrams using the Alloy analyzer with and without the UOST technique. Table 5 describes the set of benchmarks used for our comparison: the number of classes, associations, invariants, and attributes. The benchmarks "Script" was programmatically generated, in order to test large input models. Of these models, we consider the "Script" models to be the best possible scenarios for slicing (large models with many attributes and very few constraints).

Tables 6, 7, 8, 9, 10, and 11 summarize the experimental results obtained using the Alloy analyzer 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 5). The execution time is largely dependent on the defined scope, therefore, in order to analyze the efficiency of verification, the scope is limited to 7. The Alloy analyzer will examine all the examples with up to 7 objects, and try to find one that violates the property. For example, saying scope 7 means that the Alloy analyzer will check models whose top level signatures have up to 7 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 820 ms (scope 7) for the execution of the "ATM Machine" and 282161 ms (scope 7) for "Script 3". Using the UOST

Table 6. Slicing results in Alloy for the Atom-Molecule example

Before Slicing				After Slicing			
Scope	TT	ST	TT+ST	STT	SST	STT+SST	Speedup %
2	3ms	9ms	12ms	3ms	5ms	8ms	34%
3	7ms	8ms	15ms	3ms	6ms	9ms	40%
4	12ms	8ms	20ms	4ms	6ms	10ms	50%
5	17ms	10ms	27ms	4ms	9ms	13ms	52%
6	16ms	15ms	31ms	5ms	9ms	14ms	55%
7	19ms	15ms	34ms	6ms	9ms	15ms	56%

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

Table 7. Slicing results in Alloy for the University example

Before Slicing				After Slicing			
Scope	TT	ST	TT+ST	STT	SST	STT+SST	Speedup %
2	7ms	10ms	17ms	3ms	5ms	8ms	53%
3	14ms	19ms	33ms	5ms	8ms	13ms	61%
4	28ms	20ms	48ms	7ms	10ms	17ms	62%
5	36ms	31ms	67ms	12ms	15ms	27ms	65%
6	45ms	50ms	95ms	17ms	15ms	32ms	67%
7	81ms	77ms	158ms	34ms	17ms	51ms	68%

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

Table 8. Slicing results in Alloy for the 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%
5	158ms	173ms	332ms	20ms	12ms	32ms	90%
6	233ms	367ms	600ms	25ms	23ms	48ms	92%
7	325ms	495ms	820ms	30ms	28ms	58ms	93%

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

Table 9. Slicing results in Alloy for script 1

Before Slicing				After Slicing			
Scope	TT	ST	TT+ST	STT	SST	STT+SST	Speedup %
2	110ms	133ms	243ms	7ms	9ms	16ms	93%
3	161ms	290ms	451ms	9ms	9ms	18ms	96%
4	224ms	591ms	815ms	14ms	12ms	26ms	97%
5	349ms	606ms	955ms	17ms	16ms	33ms	97%
6	589ms	1077ms	1666ms	27ms	25ms	52ms	97%
7	799ms	1392ms	2191ms	38ms	25ms	63ms	97%

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

Table 10. Slicing results in Alloy for script 2

Before Slicing				After Slicing			
Scope	TT	ST	TT+ST	STT	SST	STT+SST	Speedup %
2	1839ms	3021ms	4860ms	6ms	7ms	13ms	99.7%
3	2567ms	7489ms	10056ms	11ms	8ms	19ms	99.8%
4	3374ms	8320ms	11694ms	14ms	9ms	23ms	99.8%
5	4326ms	21837ms	26163ms	18ms	14ms	32ms	99.8%
6	5231ms	32939ms	38170ms	25ms	14ms	39ms	99.8%
7	6477ms	59704ms	66181ms	35ms	16ms	51ms	99.9%

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

Table 11. Slicing results in Alloy for script 3

Before Slicing				After Slicing			
Scope	TT	ST	TT+ST	STT	SST	STT+SST	Speedup %
2	9548ms	12941ms	22489ms	6ms	8ms	14ms	99.93%
3	9734ms	30041ms	39775ms	13ms	10ms	23ms	99.94%
4	12496ms	66861ms	79357ms	19ms	10ms	29ms	99.96%
5	15702ms	85001ms	100703ms	22ms	13ms	35ms	99.96%
6	19496ms	185118ms	204614ms	29ms	16ms	45ms	99.97%
7	23089ms	259072ms	282161ms	35ms	17ms	52ms	99.98%

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

approach, it takes only 58 ms (scope 7) for "ATM Machine" and 52 ms (scope 7) for "Script 3". It is an improvement of 93% and 99.98%, respectively. In addition, the improvement can also be achieved for larger scopes as well. For instance, results for up to scope 50 can be achieved for the "ATM Machine" and scope 35 for "Script". However, without slicing we could only run the analysis for limited scopes.

6 Related Work

In this section, we discuss existing work on model partitioning or slicing. Most of the work in this area is done for UML Architectural Models, Model Slicing, and Program Slicing which is limited to slicing only. Their goal of slicing is to break larger programs or models into small submodels to reuse the required segments. However, research work on partitioning of UML/OCL models in terms of verifiability is not found in the literature. Previously, we proposed a slicing technique for models considering a UML class diagrams annotated with unrestricted OCL constraints and a specific property to verify [17]. The slicing approach was based on disjoint slicing, clustering and the removal of trivially satisfiable constraints. An implementation of the slicing technique has been developed in a UMLtoCSP tool. Experimental results demonstrate that slicing can verify complex UML/OCL models and speed-up the verification time.

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

6.1 UML Model Slicing

A theory of *model slicing* to support and maintain large UML models is mostly discussed. Current approaches of model verification have an exponential worst-case runtime. Context free slicing of the model summarizes static and structural characteristics of a UML model. The term context points towards the location of a particular object. It takes into account static and structural aspects of a UML model and excludes the enclosure of interaction information [10]. Similarly, to compute a slice of a class hierarchy of a program, it is necessary to eliminate those slices that are unnecessary thereby ensuring that the behavior of the programs would not be affected. This approach represents the criteria of model abstraction [9].

One possible approach to manage the complexity of the UML metamodel is to divide the metamodel into a set of small metamodels for each discussed UML diagram type [1]. The proposed method defines a metamodel of a directed multi-graph for a UML Metamodel Slicer. The slicer builds submetamodels for a diagram with model elements. Another slicing technique for static and dynamic UML models presents the transformation of a UML architectural model into a Model Dependency Graph (MDG). It also merges a different sequence of diagrams with relevant information available in a class diagram [13].

6.2 Architectural Slicing

The concept of *architectural slicing* is used to remove irrelevant components and connectors, so that the behavior of the slice is preserved [19]. This research introduces a new way of slicing. Architectural slicing is used to slice a specific part of a system's architecture. The sliced part is used to view higher level specifications. Similar to this approach, a dependency analysis technique is developed which is based on the slicing criteria of an architectural specification as a set of component parts [7]. The technique is named chaining. It supports the development of software architecture by eliminating unnecessary parts of the system. Furthermore, the notion of dynamic software architecture slicing (DSAS) supports software architecture analysis. This work is useful when a huge amount of components is available. DSAS extracts the useful components of the software architecture [12].

6.3 Program Slicing

Program slicing [18, 6] techniques work on the code level, decomposing source code automatically. In this research, a dataflow algorithm is presented for program slices. A recursive program written in the Pascal language is used to compute the slices. A comparable algorithm is developed to slice the hierarchies of C++ programs. It takes C++ class and inheritance relations as an input and eliminates all those data members, member functions, classes, and relationships that are irrelevant ensuring that the program behavior is maintained. This work gave us the motivation to reduce and eliminate those classes and relationships which do not have any relation to the UML/OCL model [6].

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

7 Conclusion and Future Work

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

As part of our future work, we plan to explore three research directions. First, we plan to optimize our slicing approach by eliminating loosely coupled superclasses and subclasses. Second, we optimize our UOST by discarding aggregations and compositions with optional multiplicities. On the other hand, we also plan to explore different sets of values for multiplicities such as a bus may have [45, 55, 65] seats instead of 1..* seats. Third, we will investigate a feedback technique that provides useful directions to a software engineer in case of unsatisfiability to allow the software engineer to focus her attention to the incorrect submodels while ignoring the rest of the model.

References

1. Bae, J.H., Lee, K., Chae, H.S.: Modularization of the UML Metamodel Using Model Slicing. In: ITNG, pp. 1253–1254. IEEE Computer Society, Los Alamitos (2008)
2. Cabot, J., Clarisó, R.: UML/OCL Verification in Practice. In: MoDELS 2008. Workshop on Challenges in MDE, ChaMDE 2008 (2008)
3. Cabot, J., Clariso, R., Riera, D.: Papers and Researchers: An Example of an Unsatisfiable UML/OCL Model, http://gres.uoc.edu/UMLtoCSP/examples/Papers-Researchers.pdf
4. Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: A Tool for the Formal Verification of UML/OCL Models Using Constraint Programming. In: ASE 2007, pp. 547–548. ACM, New York (2007)
5. Cabot, J., Clarisó, R., Riera, D.: Verification of UML/OCL Class Diagrams Using Constraint Programming. In: ICSTW 2008, pp. 73–80. IEEE Computer Society, Los Alamitos (2008)
6. Georg, G., Bieman, J., France, R.B.: Using Alloy and UML/OCL to Specify Runtime Configuration Management: A case study. In: Workshop of the UML-Group
7. Stafford, D.J.R.J.A., Wolf, A.L.: Chaining: A Software Architecture Dependence Analysis Technique. Technical Report, University of Colorado, Department of Computer Science (1997)
8. Jackson, D.: Alloy: A Lightweight Object Modelling Notation. ACM Transactions on Software Engineering and Methodology 11(2), 256–290 (2002)
9. Choi, J.-D., Field, J.H., Ramalingam, G., Tip, F.: Method and Apparatus for Slicing Class Hierarchies, http://www.patentstorm.us/patents/6179491.html
10. Kagdi, H.H., Maletic, J.I., Sutton, A.: Context-free Slicing of UML Class Models. In: ICSM 2005, pp. 635–638. IEEE Computer Society, Los Alamitos (2005)
11. Kellomäki, P.: Verification of Reactive Systems Using DisCo and PVS. In: Fitzgerald, J.S., Jones, C.B., Lucas, P. (eds.) FME 1997. LNCS, vol. 1313, pp. 589–604. Springer, Heidelberg (1997)
12. Kim, T., Song, Y.-T., Chung, L., Huynh, D.T.: Dynamic Software Architecture Slicing. In: COMPSAC, pp. 61–66. IEEE Computer Society, Los Alamitos (1999)
13. Lallchandani, J.T., Mall, R.: Slicing UML Architectural Models. In: ACM / SIGSOFT SEN, vol. 33, pp. 1–9 (2008)
14. Lanubile, F., Visaggio, G.: Extracting Reusable Functions by Flow Graph-based Program Slicing. IEEE Trans. Softw. Eng. 23(4), 246–259 (1997)
15. Ojala, V.: A Slicer for UML State Machines. Technical Report 25, Helsinki University of Technology (2007)
16. Qi Lu, J.Q., Zhang, F.: Program Slicing: Its Improved Algorithm and Application in Verification. Journal of Computer Science and Technology 3, 29–39 (1988)
17. Shaikh, A., Clarisó, R., Wiil, U.K., Memon, N.: Verification-driven Slicing of UML/OCL Models. In: ASE, pp. 185–194 (2010)
18. Weiser.sk, M.: Program Slicing. IEEE Trans. Software Eng., 352–357 (1984)
19. Zhao, J.: Applying Alicing Technique to Software Architectures. CoRR, cs.SE/0105008 (2001)