Using Slicing to Improve the Performance of Model Invariant Checking

Wuliang Sun, Benoit Combemale, Robert B. France, Arnaud Blouin, Benoit Baudry, Indrakshi Ray

▶ **To cite this version:**

Wuliang Sun, Benoit Combemale, Robert B. France, Arnaud Blouin, Benoit Baudry, et al.. Using Slicing to Improve the Performance of Model Invariant Checking. The Journal of Object Technology, 2015, pp.28. 10.5381/jot.2015.14.4.a1 . hal-01179369

HAL Id: hal-01179369

https://inria.hal.science/hal-01179369v1

Submitted on 22 Jul 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NoDerivatives 4.0 International License

JOURNAL OF OBJECT TECHNOLOGY
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

Using Slicing to Improve the Performance of Model Invariant Checking

Wuliang Sun[a] Benoit Combemale[bc] Robert B. France[a]

Arnaud Blouin[cd] Benoit Baudry[c] Indrakshi Ray[a]

a. Colorado State University, Fort Collins, Colorado, USA
b. University of Rennes 1, Rennes, France
c. INRIA, Rennes, France
d. INSA, Rennes, France

Abstract In Model Driven Development (MDD), it is important to ensure that a model conforms to the invariants defined in the metamodel. Such invariant checking can improve developers' understanding of modeled aspects of complex systems and uncover structural errors in design models. General-purpose rigorous analysis tools that check invariants are likely to perform the analysis over the entire metamodel and model. Since modern day software is exceedingly complex, the size of the model together with the metamodel can be very large. Consequently, invariant checking can take a very long time. For example, checking a model consisting of 5,000 elements can take up to several hours if the analysis completes. Moreover, sometimes the analysis process cannot be completed as the system resources get exhausted. To this end, we introduce model slicing within the invariant checking process, and use a slicing technique to reduce the size of the inputs in order to make invariant checking of large models feasible with existing tools. The evaluation we performed provides evidence that model slicing can significantly reduce the time to perform the invariant checking. In the experiments that we conducted, we achieved speedups ranging from 1.5 to 36.0 and we also demonstrate the correctness of the checking results.

Keywords UML, Metamodel, Model, OCL, Invariant Checking, Model Slicing

1 Introduction

Model-driven development (MDD) is a paradigm that (1) promotes models as the major artifacts to drive the software development process, and (2) uses model transformation and code

Wuliang Sun, Benoit Combemale, Robert B. France, Arnaud Blouin, Benoit Baudry, Indrakshi Ray. *Using Slicing to Improve the Performance of Model Invariant Checking*. In Journal of Object Technology, vol. 0, 2015, pages 0:1–28. Available at http://www.jot.fm

generation to bridge the gap between high-level design models and low-level implementations. In MDD, models are often used by code generators. Since design errors in the models may be propagated into the implementations via model-to-code transformation, it is very important to uncover design errors during the early stages of software development.

In MDD, models must conform to the well-formedness rules of the metamodel. Such well-formedness rules can be thought of as invariants of the metamodel. One needs to check the models to ensure that the invariants of the metamodel are satisfied using automated tools such as Eclipse OCL [Tea05], so that the developers can identify potential problems during design time before they are used to generate code. However, the existing tools are inefficient for invariant checking on large models. For example, as shown in experiments we conducted and described in this paper, checking model instances consisting of hundreds of thousands of elements against a metamodel that includes 345 elements would take more than two hours. Thus, there is a need for techniques that support invariant checking for large models and metamodels.

Slicing techniques [Wei81] produce reduced forms of artifacts that can be used to support, for example, analysis of artifact properties. Slicing techniques have been proposed for different software artifacts, including programs (e.g., see [GL91][Wei81]), and models (e.g., see [ABC$^+$11][BCBB12][EW04][KMS05][KSTV03]). In the MDD area, model slicing techniques have been used to support a variety of modeling tasks, including model comprehension [ABC$^+$11][BCBB12][KSTV03], analysis [JGB11][LKR10][LKR11], and verification [EW04][SCWM10][SWM11].

In model slicing techniques, *slicing criteria* are input data used to determine the elements that are included in slices. Model slicing techniques typically proceed in two steps: (1) The dependency between model elements of interest (e.g., elements satisfying a *slicing criterion*) and the rest of the model is analyzed using heuristics related to a model's properties (e.g., the structure of a model); and (2) a fragment of the model consisting only of elements satisfying a slicing criterion, is extracted from the model.

In this paper we introduce the model slicing technique to the invariant analysis process. The approach aims to improve the size of the model that can be checked using invariant checking tools. The approach is not intended to improve the existing invariant checking algorithms. Instead, the approach aims to reduce the size of the checking inputs to make the analysis more efficient. It means our approach preprocesses the input of the invariant checking process, and thus is agnostic to the checking technologies the software developers are working with. In this paper we focus on analyses that involve checking the consistency between a model and the invariants defined in a metamodel. However, checking whether a model is a valid instance of the metamodel is out of scope of the paper. This means that we assume the model conforms to the structural constraints (e.g., multiplicity constraints) defined in the metamodel, but may or may not satisfy the invariants defined in the metamodel.

We have developed a framework that provides: (1) an implementation of the model slicing technique; (2) an implementation for checking models against invariants defined in the metamodels. The framework was implemented using Java and the Eclipse Modeling Framework (EMF) [SBMP09]. Even though the evaluation framework builds upon Java and Eclipse, the slicing technique is not bound to a particular technological space, and it can be implemented using any language and framework. We have evaluated our technique to check whether (1) the slicing improves the efficiency of the invariant checking, and (2) the invariant checking results for the sliced models are the same as the unsliced models. We have evaluated our approach with the Java metamodel and 73 models produced by reverse engineering Eclipse plugins. The evaluation we performed provides evidence that the proposed slicing technique can significantly reduce the time to perform the invariant checking while preserving the

checking results. We also show that the invariant checking approach described in the paper can offer similar performance gains on small manually built models (e.g. hundreds of elements).

The rest of the paper is organized as follows. Section 2 provides background material needed to understand the work described in the paper. Section 3 presents the big picture of the invariant analysis approach. Section 4 describes the approach in details, and Section 5 presents the results of an evaluation of the approach. Section 6 provides a discussion of the approach. Section 7 describes the related work, and Section 8 concludes the paper.

2 Background

In this section, we provide background material needed to understand the proposed invariant analysis approach described in the paper. Section 2.1 describes a list of tools that can be used for invariant checking, and our invariant analysis approach builds upon one of these tools. Section 2.2 describes a variety of model slicing techniques. We leverage on these works to apply the slicing technique to the invariant analysis that involves both the metamodel and the model, which implies a particular co-slicing approach.

2.1 Invariant Checking

USE [GBR07], developed by the Database Systems Group at Bremen University, is a modeling tool for specifying object-oriented systems. It allows developers to generate models (expressed using the object diagram notation) that are checked against user-specified properties such as invariants expressed in a metamodel (expressed using the class diagram notation). A system with a set of invariants can be specified using the USE specification language, that is based on a subset of the Unified Modeling Language (UML) [EFLR99] and the Object Constraint Language (OCL) [Spe07a]. A model can be created using the shell commands provided by the USE tool. The feedback provided by the USE tool includes highlighted invariants that are inconsistent with the given model.

Alloy [Jac02] is a formal specification language that was developed by the Software Design Group at MIT. It has good tool support in the form of the Alloy Analyzer that translates an Alloy specification into a boolean formula that is evaluated by embedded SAT-solvers. The Alloy Analyzer generates examples or counter-examples of certain properties by exploring a search space given by limiting the number of entities in the Alloy model. An Alloy model consists of signature declarations, fields, facts, and predicates. Each field belongs to a signature and represents a relation between two or more signatures. Facts are statements that define constraints on the elements of the model. Predicates are parameterized constraints that can be invoked from within facts or other predicates. The Alloy Analyzer allows developers to specify (1) metamodels using signatures, and fields, (2) models using predicates, and (3) invariants using facts, and returns information showing whether models satisfy the invariants.

The Kermeta language [JCB+13] was developed by the Triskell Team at INRIA. It is an executable metamodeling language implemented on top of the Eclipse Modeling Framework (EMF) [SBMP09] within the Eclipse development environment. It has been used for specifying models, and model transformations that are compliant to the Meta Object Facility (MOF) standard [Omg08]. The Kermeta workbench allows developers to specify well-formedness rules, often formulated as invariants, on metamodels. These rules can be expressed using an OCL-like specification language. The Kermeta workbench provides several APIs for evaluating OCL-like invariants against models. It reports warning information if a given model does not satisfy the invariants defined in the metamodel.

The Eclipse OCL project [Tea05] is an implementation of the OCL standard [Spe07a] for EMF-based models. It provides APIs for (1) analyzing and transforming the abstract syntax model of OCL expressions, and (2) parsing and evaluating invariants and queries on metamodels. The extensibility of the provided APIs allows software modelers to develop their own customized prototypes for a variety of invariant checking tasks. The invariant checking approach described in the paper builds upon the Eclipse OCL project.

2.2 Model Slicing

Kagdi et al. [KMS05] proposed an approach for slicing class models. The slices are applicable to models that do not require a context (e.g., a set of scenarios in which objects are involved) for the computation of a model slice. OCL invariants are not considered in their slicing approach.

Blouin et al. [BCBB11] described a model-driven approach to specifying model slicers for different domain specific modeling languages. The approach relies on *Kompren*, a modeling language dedicated to the construction of model slicers, that aims at automatically building model slicers for any languages. *Kompren* can be used to produce a slicer for the slicing approach described in the paper.

Sen et al. [SMBJ09] proposed a slicing technique for metamodel pruning by removing unnecessary classes and properties from a metamodel. The slicing technique takes as input a large metamodel and a slicing criterion including a set of classes and properties of interest, and produces a pruned metamodel that is a subset of the input metamodel. The pruned metamodel contains all the model elements specified in the slicing criterion. Any instance of the pruned metamodel is also an instance of the input metamodel.

Jeanneret et al. [JGB11] used the slicing technique to estimate the part of a model used by an operation without executing the operation. The slicing technique takes as input a model, an operation and a metamodel in which the operation is defined, and produces a footprint of the operation which consists of the set of metamodel elements involved in the operation contract. The model elements that are the instances of the metamodel elements involved in the footprint would be used by the operation.

Lano et al. [LKR10][LKR11] described an approach to slicing a class with operation specifications and invariants. Their approach uses a state machine to specify a sequence of operation invocations on an instance of a class. If a class feature, such as an attribute, does not occur in any operation defined in the class, it can be removed together with any invariants that refer to it. Compared to their approach, our slicing approach focuses on slicing a metamodel and its models, and does not require a state machine to guide the slicing process.

In our prior work, we proposed a slicing technique for metamodels that include OCL invariants and operation specifications [SFR13]. The slicing technique is used to improve the efficiency of a model analysis technique that involves checking a sequence of operation invocations to uncover violations in specified invariants [SFR11]. The slicing approach automatically generates slicing criteria consisting of a subset of invariants and operation specifications, and uses the criteria to extract metamodel fragments, where each metamodel fragment can be analyzed separately. Unlike the prior work, the slicing technique described in the paper is used to improve the efficiency of the invariant checking. Thus it uses an invariant as a slicing criterion to slice the metamodel and model, and does not deal with operation specifications.

3 Overview of Our Approach

In this section we motivate the need for slicing when checking invariants (Section 3.1), we introduce a motivating example to illustrate the role of the slicing technique in the context of invariant checking (Section 3.2), and describe the approach overview (Section 3.3).

3.1 Motivation

Errors detected in the design phase of the software development process are less expensive to fix than those detected later. One approach used by software designers for checking correctness is to ensure that a model conforms to a predefined metamodel using automated tools. This involves the following steps. (i) A software modeler designs a metamodel and defines a set of Well-Formedness Rules (WFRs) specified in the form of invariants which must be satisfied by models conforming to the metamodel. (ii) He creates a set of models that conform to the metamodel structure. (iii) He checks models against the WFRs using invariant checking tools such as the Eclipse OCL checker [Tea05]. The entire process works well for small models with hundreds of elements, and the modeler can receive the feedback from the checking tools within seconds or minutes.

However, given the growing complexity of software systems, models used to represent these complex systems are also growing significantly in size. While design models were built by hand in the early days of MDD, nowadays models with possibly more than one million elements can be built programmatically. For example, we used a reverse engineering tool, namely MoDisco [BCDM14], to generate Java models from multiple Eclipse platform plugins. These models can have up to one million elements. The checking time goes up significantly for large models with hundreds of thousands of elements. This motivates the improvement of the scalability of the invariant checking approach in the context of large models.

In the approach described in the paper, checking models against invariants does not need the entire metamodel and the full model to be actually processed. Only a small part of the metamodel and model that are referenced by the invariants needs to be used for the invariant checking. This is due to the fact that a substantial number of invariants only reference part of the metamodel in which they are defined [CCB12]. This motivates the use of the slicing technique in the context of invariant checking. The slicing technique thus can be used to reduce the size of the input metamodel and model to make the checking more efficient.

3.2 Motivating Example

Figure 1 shows part of a metamodel that describes the information system of a bus company (excerpted from [SCWM10]). In the metamodel, a trip uses more than one coach. A coach is controlled by multiple security guards. A passenger can buy multiple tickets from a vending machine located in a booking office. Adult and child tickets are available for sale. A passenger can select more than one trip, where each trip can be either private or regular. A booking office is managed by at most one manager. Figure 2 shows a valid instance of the metamodel given in Figure 1.

Invariants defined in the metamodel are given in Table 1. For example, the *UniqueTicket-Number* invariant is defined in the context of the *Ticket* class in the metamodel, and it specifies that every ticket must have a unique number. Such invariants cannot be expressed directly using the class diagram notation, and thus are specified using other languages such as the OCL [Spe07a]. The model in Figure 2 may or may not satisfy the invariants defined in the metamodel.

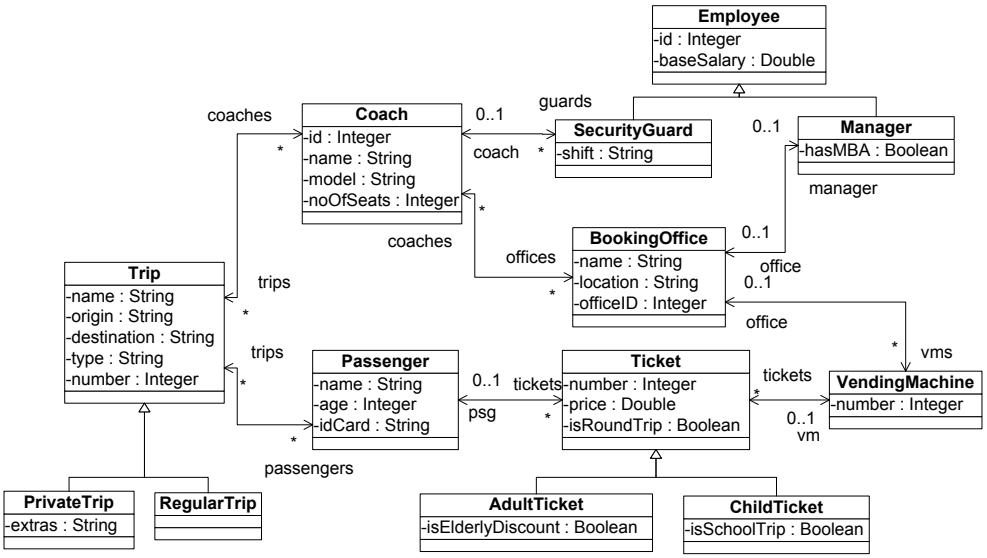


Figure 1 – A metamodel expressed using a class diagram describing the information system of a bus company

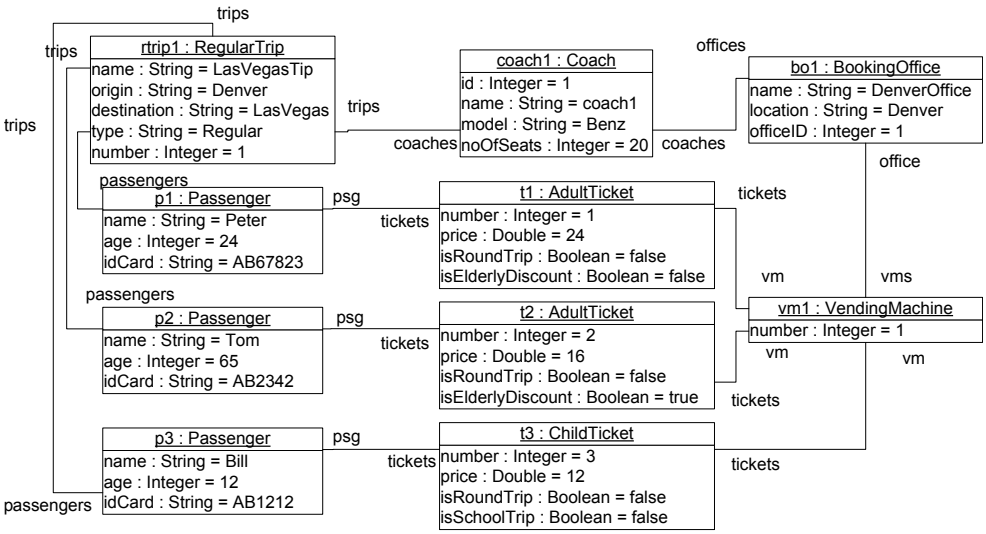


Figure 2 – A model expressed using an object diagram conforming to the metamodel given in Figure 1

Tools such as the Eclipse OCL checker [Tea05] can be used in this situation for invariant checking. They take as input the invariant, the metamodel and the model, and produce checking results, indicating whether the model is consistent with the invariant defined in the metamodel. In this case, the model in Figure 2 does not violate the *UniqueTicketNumber* invariant defined in the metamodel since *t1, t2, t3* have different numbers (i.e., 1, 2, 3).

Note that to check the *UniqueTicketNumber* invariant, we only need to look into the tickets and their numbers. The rest of the model is not relevant to the checking. Indeed, the *UniqueTicketNumber* invariant is defined in the context of the *Ticket* class and only refers to the *number* attribute in the *Ticket* class. Therefore, instead of feeding the entire metamodel in

Table 1 – A list of invariants in the metamodel

// Each coach has more than ten seats. **Context** *Coach* **inv MinCoachSize:** self.noOfSeats $\geq$ 10
// For every trip *t* that is assigned to a coach, the number of passengers associated // with *t* must be smaller than the number of seats allowed for a coach. **Context** *Coach* **inv MaxCoachSize:** self.trips→forAll(t | t.passengers→size()$\leq$ noOfSeats)
// Each ticket must have a unique number. **Context** *Ticket* **inv UniqueTicketNumber:** Ticket::allInstances()→forAll(t1, t2 | t1.number = t2.number implies t1 = t2)
// Each regular trip must have more than six passengers. **Context** *RegularTrip* **inv MinPassengers:** self.passengers→size() $\geq$ 6

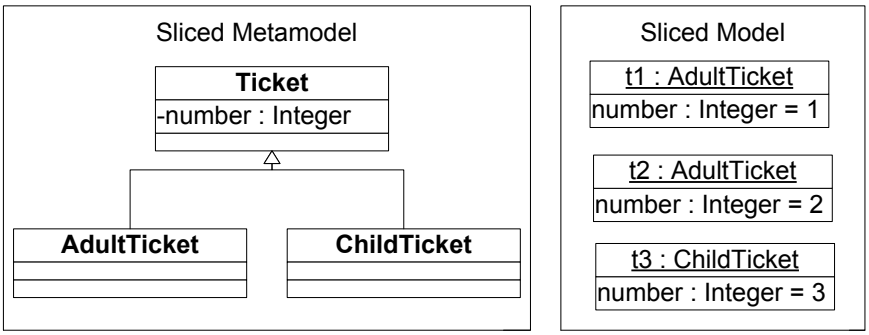


Figure 3 – Checking the *UniqueTicketNumber* invariant in the context of sliced metamodel and model

Figure 1 and the full model in Figure 2 into the invariant checking tools, we can use the slicing technique to generate a smaller metamodel and model. Figure 3 shows the sliced metamodel and model that are parts of the original metamodel and model. We can feed these generated metamodel and model into the tools for invariant checking. Note that we keep the classes *AdultTicket* and *ChildTicket* in the generated metamodel since *UniqueTicketNumber* is defined in the context of the *Ticket* class and thus can be inherited by the subclasses of the *Ticket* class. Therefore the instances of *AdultTicket* and *ChildTicket* also need to be checked against the *UniqueTicketNumber* invariant.

It is also important to note that the slicing technique is needed for invariant checking only if (1) the use of the slicing technique can reduce the invariant checking time (i.e., effectiveness of the slicing technique) and (2) checking the sliced model produces the same results as those generated by verifying the invariants against the original model (i.e., the correctness of the slicing technique). For example, the slicing technique is needed for checking the *UniqueTicketNumber* invariant if (1) the checking time for the metamodel and model in Figure 3 is less than that used for original metamodel and model, and (2) the invariant checking results for the original models are the same as the results for the models in Figure 3. We have conducted an evaluation to explore the effectiveness and the correctness of the slicing technique in the context of invariant checking, and the evaluation results are given in the Evaluation Section.

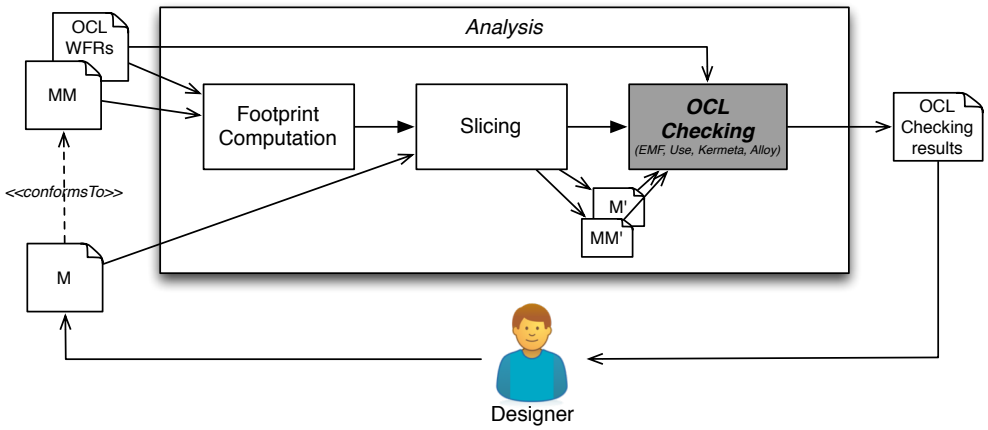


Figure 4 – Approach overview

Table 2 – Footprint (i.e., dependent elements) of each invariant (see Table 1) defined in the metamodel in Figure 1

Invariant	**Dependent Classes**	**Dependent Attrs/Refs**
MinCoachSize	Coach	noOfSeats
MaxCoachSize	Coach, Trip, Passenger	trips, passengers, noOfSeats
UniqueTicketNumber	Ticket	number
MinPassengers	RegularTrip, Trip, Passenger	passengers

3.3 Approach Overview

Figure 4 shows an overview of the proposed invariant analysis approach. The input of the checking includes a metamodel (*MM*), a model (*M*), and one or many OCL invariants (*Well-Formedness Rules*). First, the approach computes a footprint of the OCL invariants on the metamodel. A footprint refers to part of a metamodel that contains all elements that affect the outcome of an operation [JGB11]. In this paper a footprint refers to all metamodel elements that are directly referenced by the input OCL invariants. Second, the footprint serves as slicing criterion, and is used to generate a sliced metamodel (*MM'*) from the input metamodel. The sliced metamodel (*MM'*) includes (1) all the metamodel elements from the footprint, and (2) all the subclasses of the classes in the footprint. Third, the sliced metamodel (*MM'*) is used to generate a sliced model (*M'*) from the input model. The sliced model (*M'*) contains only model elements that are instances of metamodel elements in *MM'*. Finally, the sliced metamodel and model with the invariants are fed into the tools for invariant checking.

4 Our Detailed Approach

In this section we illustrate the invariant checking approach in details.

4.1 Generating Footprint

Table 2 shows the footprint of each invariant given in Table 1. The footprint of an invariant contains metamodel elements such as classes, attributes, references, and/or enumerations. The

Algorithm 1 Generate a footprint

```
1: Input: A metamodel MM and an invariant Inv
2: Output: A footprint FP
3: Algorithm Steps:
4: Set FP = Inv's context class;
5: for each association end call, Aec in Inv do
6:    if Aec corresponds to an element Elmt (i.e., attribute or reference) in MM then
7:       FP = FP ∪ Elmt;
8:       FP = FP ∪ Elmt's type class;
9:    end if
10: end for
11: Return FP;
```

footprint computation takes a metamodel and an invariant, and analyzes the dependencies between the invariant and the metamodel. The dependency analysis is performed by traversing the syntax tree of the OCL invariant.

Algorithm 1 is used to generate a footprint (*FP*) from a metamodel (*MM*) and an invariant *Inv*. For example, consider the footprint computation of the *MaxCoachSize* invariant. The *MaxCoachSize* invariant is defined in the context of class *Coach*, and thus depends on class *Coach*. The expression $self.trips$ is an association end call expression and it returns a set of trips assigned to the coach (referred to by $self$). There is thus a dependency with reference $trips$, and its type class, $Trip$ via the class *Coach*. The parameter t in the *MaxCoachSize* invariant refers to an instance of class $Trip$, and the expression $t.passengers$ returns a set of passengers associated with a trip. There is thus a dependency with reference *passengers*, and its type class, *Passenger* via the class $Trip$. The expression $noOfSeats$ refers to an attribute defined in class *Coach*, and the invariant thus depends on attribute $noOfSeats$ and its containing class, *Coach*. The footprint computation thus reveals the elements the invariant refers to and thus depends on the following metamodel elements: *Coach*, $Trip$, *Passenger*, $trips$, *passengers* and $noOfSeats$.

Note that the *MinPassengers* invariant depends on both *Trip* and *Passenger* classes. This is because *MinPassengers* uses the *passengers* reference in its definition (see Table 1). Reference *passengers* is defined in the context of class *Trip* and can be inherited by the subclasses (e.g., *RegularTrip*) of *Trip*. In addition, the type of the *passengers* reference is class *Passenger*.

4.2 Slicing Metamodel

Algorithm 2 is used to generate a sliced metamodel (*MM'*) from a footprint (*FP*) and the original metamodel (*MM*). The slicing criteria in this case would be the metamodel elements in the footprint. Algorithm 2 computes *Subs*, a set of classes that are the subclasses of the classes in the footprint. The sliced metamodel contains only elements that are from the footprint and *Subs*. The reason we keep these referred classes' subclasses in the sliced metamodel is that the instances of subclasses are also the instances of their super classes, and thus can be used for invariant checking.

Consider the case in which a modeler wants to check whether the model in Figure 2 satisfies the *MaxCoachSize* invariant defined in the metamodel in Figure 1. Algorithm 2 can be used in this case to slice the metamodel in Figure 1 using the footprint of the *MaxCoachSize* invariant. Figure 5 shows the sliced metamodel that is generated from the footprint of the *MaxCoachSize* invariant. Since the footprint of the *MaxCoachSize* invariant includes class *Trip*,

Algorithm 2 Slice a metamodel

```
1: Input: A footprint FP and the metamodel MM
2: Output: A sliced metamodel MM'
3: Algorithm Steps:
4: Set Subs = {};
5: for each element, Elmt, in FP do
6:    if Elmt is a class then
7:       for each indirect and direct subclass, Sub, of Elmt do
8:          Subs = Subs ∪ Sub;
9:       end for
10:   end if
11: end for
12: Return FP ∪ Subs;
```

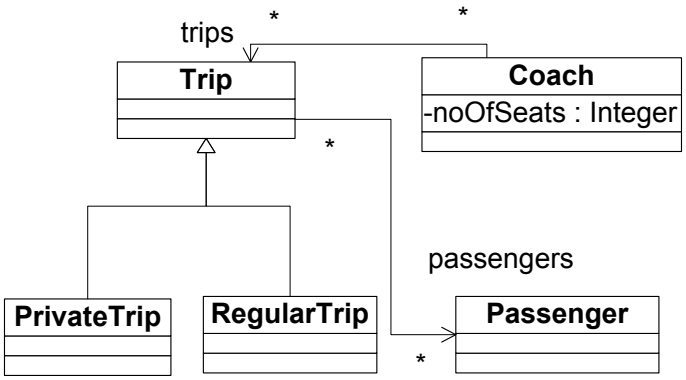


Figure 5 – A metamodel slice generated from the footprint of the *MaxCoachSize* invariant

and class *Trip* has two subclasses, *PrivateTrip* and *RegularTrip*, the *MaxCoachSize* invariant also depends on *PrivateTrip* and *RegularTrip*. In summary the metamodel elements that are referenced by the *MaxCoachSize* invariant include *Coach*, *Trip*, *PrivateTrip*, *RegularTrip*, *Passenger*, *trips*, *passengers*, and *noOfSeats*.

4.3 Slicing Model

Algorithm 3 is used to slice a model. It takes as input a model (*M*) and a sliced metamodel (*MM'*), and produces a sliced model, where each element in the sliced model is an instance of a metamodel element in the sliced metamodel. For example, given the sliced metamodel in Figure 5 and the model in Figure 2, Algorithm 3 can be used to generate a sliced model, shown in Figure 6, that conforms to the sliced metamodel in Figure 5. Note that the sliced model is a valid instance of the sliced metamodel, but it may or may not satisfy the invariants.

The algorithm checks each object (see lines 5-7) in *M*, and removes an object and its slots/link ends if the object's metaclass is not in the set of classes involved in *MM'*. For example, object *bo1:BookingOffice* in Figure 2 is not an instance of any class involved in the sliced metamodel in Figure 5, and thus can be removed from the model. In addition, its slots (i.e. name, location, and officeID) and link ends (i.e. coaches and vms) are also removed from the model. Note that both slots and link ends are the modeling concepts used in the object

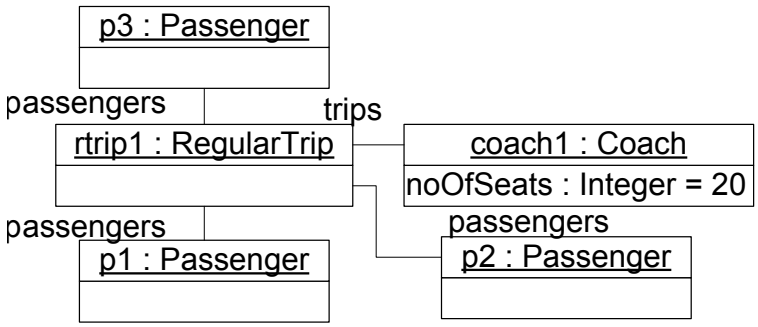


Figure 6 – An example of a sliced model

Algorithm 3 Slice a model

```
1:  Input: A model, M, and a sliced metamodel, MM'
2:  Output: A sliced model that conforms to MM'
3:  Algorithm Steps:
4:  Set Clss = a set of classes in MM', set Attrs = a set of attributes in MM', set Refs = a set of
    references in MM';
5:  for each object, obj, in M do
6:     if obj's metaclass not in Clss then
7:        Remove obj and its slots/link ends from M;
8:     else
9:        for each slot, sl, of obj do
10:          if sl's corresponding attribute not in Attrs then
11:             Remove sl from obj;
12:          end if
13:       end for
14:       for each link end, le, of obj do
15:          if le's corresponding reference not in Refs then
16:             Remove le from obj;
17:          end if
18:       end for
19:    end if
20: end for
21: Return M;
```

diagram notation, where slots are the instances of attributes and link ends are the instances of references.

Algorithm 3 also checks the slots and link ends of each unremoved object. Lines 9-13 remove a slot of an object if the slot's corresponding attribute is not included in *Attrs*, a set of attributes in *MM'*. Lines 14-18 remove a link end of an object if the link end's corresponding reference is not included in *Refs*. For example, object *rtrip1:RegularTrip* in Figure 2 has 5 slots and 4 link ends. Since class *RegularTrip* and its super class *Trip* in Figure 5 have no attributes, the slicing algorithm removed all the slots from object *rtrip1:RegularTrip* as indicated in Figure 6. Similarly, class *RegularTrip* and its super class *Trip* in Figure 5 have only one reference, *passengers*. Thus, the slicing algorithm removed link end *coaches* from object *rtrip1:RegularTrip*.

4.4 OCL Checking

In summary, the slicing technique uses the *MaxCoachSize* invariant to generate a sliced metamodel (see Figure 5) and a sliced model (see Figure 6) from the input metamodel (see Figure 1) and model (see Figure 2). The invariant and the sliced metamodel and model can be fed into the OCL checking tool for invariant analysis. The checking tool would return an analysis result, indicating whether the sliced model satisfies the invariant in the context of the sliced metamodel.

5 Evaluation

In this section we evaluate the effectiveness and correctness of the proposed slicing technique. Specifically the evaluation aims to answer the following research questions:

RQ1 Can the slicing technique significantly improve the efficiency of the invariant checking?

RQ2 Is the slicing technique ensured to preserve the invariant checking results?

To answer *RQ1*, we need to check whether the invariant checking time for unsliced metamodel and model (*CTUM*) is greater than the invariant checking time for sliced metamodel and model (*CTSM*). To avoid the bias of simply comparing the checking time of large and small models (i.e., unsliced and sliced models), we also need to take the time used to generate footprint and to slice the metamodel and model (*ST*) into consideration.

Metric 1 below can be used to answer *RQ1*:

$$CheckingTimeSpeedup(CTS) = \frac{CTUM}{CTSM + ST} \tag{1}$$

If *CTS* is greater than 1.0, the slicing technique can improve the efficiency of the invariant checking.

The slicing technique aims to improve the efficiency of the invariant checking. Thus it should not change the checking results. For example, given an invariant, a metamodel and a model, if the OCL tool returns an analysis result, indicating the model does not satisfy the invariant defined in the metamodel, the OCL tool should return the same result for the sliced metamodel and model (i.e., the sliced model does not satisfy the invariant in the context of the sliced metamodel). To answer *RQ2*, we need to check whether the invariant checking results for unsliced metamodel and model are the same as that for sliced metamodel and model.

In the remainder of this section we describe the prototype we developed for the evaluation, the data used in the evaluation, the evaluation results, and the threats to validity we identified.

5.1 Evaluation Framework Implementation

We developed an evaluation framework that provides (1) implementation of the proposed slicing technique, and (2) implementation for checking models against metamodels with invariants. The metamodels used in the evaluation are expressed using the Ecore [SBMP09] standard (i.e., the de-facto standard to define metamodels), the models used in the evaluation are expressed using the XMI [Spe07b] standard (i.e., the de-facto standard to serialize models), and the invariants used in the evaluation are expressed using the OCL [Spe07a].

The evaluation framework was implemented using Java and Eclipse development platform. Even though the evaluation framework builds upon Java and Eclipse, the slicing technique is not bound to a particular technical space, and it can be implemented using any language

and framework. The framework also builds upon the Eclipse Modeling Framework (EMF) [SBMP09] and the Eclipse OCL project [Tea05]. The framework uses the Eclipse OCL project to (1) parse the OCL invariants defined in a metamodel and (2) check a model against the metamodel with OCL invariants since the Eclipse OCL project also builds upon EMF. Both the implementations of the slicing technique and the evaluation framework can be found in *https://github.com/sunwuliang/SlicingProject3.0*.

5.2 Data Collection

The proposed slicing technique was evaluated in the context of both large and small models. The large models were automatically generated from Eclipse plugins (Java programs) using reverse engineering tools. Their sizes vary from 175926 to one million. The small models were manually built domain models, and their sizes vary from 163 to 7558. In the remainder of this section we describe the models used in the evaluation.

5.2.1 Large Models

The metamodel used for the evaluation is the Java metamodel from the EMF [SBMP09]. The reasons for this choice are: (1) it is fairly complex having 345 model elements including classes, attributes, references, and enumerations and (2) its relevant conforming models are relatively easy to collect from Java programs using reverse engineering tools. The models used for the evaluation are generated from 73 Eclipse plugins (Java programs). The reason we chose Eclipse plugins is that the models generated from the Eclipse plugins are quite large. We used a reverse engineering tool, namely MoDisco [BCDM14], to generate models from these plugins. Six invariants were used in the evaluation. The complete data used in the evaluation can be found in *https://github.com/sunwuliang/SlicingProject3.0*.

Figure 7 shows the sizes of the models used in the evaluation. The sizes of these 73 models range from 175926 (the model generated from the *org.eclipse.gmf.runtime.draw2-d.ui.render.awt* plugin) to 993319 (model generated from the *org.eclipse.emf.ecore.x-core.ui* plugin) model elements including objects, links, and slots.

5.2.2 Small Models

The small models used in the evaluation were collected from state-of-the-art model repositories: *ReMoDD* [Tea13b] and *Metamodel Zoos* [Tea13a]. Below is a list of the collected models:

- FSM [Tea13a], a domain model that describes the concepts of a finite state machine;
- ER2MOF [Tea13b], a domain model that describes a model transformation from an entity-relationship schema to a relational model;
- HTML [Tea13a], a domain model that describes the HyperText Markup Language (HTML);
- MATLAB [Tea13a], a domain model that describes the MATLAB language;
- MARTE [Tea13a], a domain model that describes the MARTE profile.

These models conform to the UML2 standard, and were manually created by domain experts. These models are much smaller than the models generated by the reverse engineering tools. The sizes of these models are given below: FSM (163), ER2MOF (779), HTML (853), MATLAB (1278), and MARTE (7558). Six invariants were used in the evaluation.

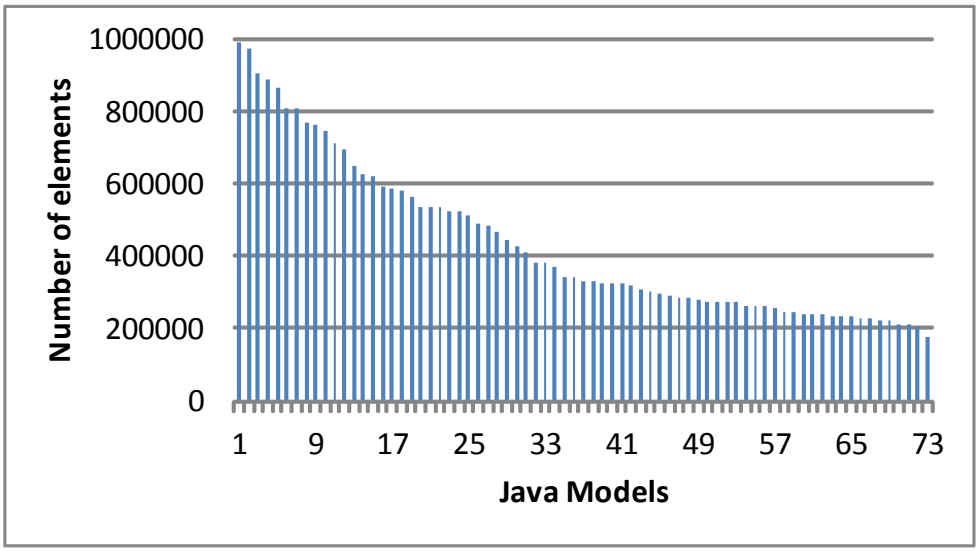


Figure 7 – The sizes of the models used in the evaluation

5.3 Evaluation Results

The evaluation was performed on a laptop computer with 2.17 GHz Intel Dual Core CPU, 3 GB RAM, and Windows 7.

5.3.1 Effectiveness of the Slicing Technique

The evaluation framework takes as input the Java metamodel, 73 models, and six invariants. For a pair of one model and one invariant, the evaluation framework performs the following four steps. First, it checks the model against the invariant in the context of Java metamodel, and measures the checking time (*CTUM*). Second, it generates a sliced Java metamodel for the invariant, slices the input model using the sliced Java metamodel, and measures the time used to generate footprint and to slice the metamodel and model (*ST*). Third, it checks the sliced model against the invariant in the context of the sliced metamodel, and measures the checking time (*CTSM*). Fourth, it calculates the *CTS* based on Metric 1. In total, there are 73 *CTS*s for each invariant. Note that to ensure the evaluation results are reliable, we calculated the CTS ten times for each pair of invariant and model, and used its average value.

Figure 8 shows the distribution of the checking time for unsliced metamodel and model for each invariant. Although most *CTUM*s are less than 2000 seconds, the *CTUM* could achieve 7860 seconds (i.e., more than two hours) in the worst case scenario (see the box plot for *Inv6*). Figure 9 shows the distribution of the checking time for sliced metamodel and model for each invariant. The checking time varies from 0.5 second to 993 seconds. Compared with the *CTUM* and the *CTSM*, the time used to generate footprints and to slice the metamodel and model (*ST*) is quite small. Figure 10 shows the distribution of the *ST*s for each invariant. All the *ST*s are less than two seconds.

Figure 11 shows the distribution of the checking time speedup for each invariant. All the *CTSs* are calculated using Metric 1. For example, given *Inv5*, the *CTS*s for the 73 models vary from 2.5 (minimum CTS) to 31.1 (maximum CTS). Since the *CTS* for each pair of invariant and model is above 1.0, the value of $CTSM + ST$ must be smaller than *CTUM* (refer to Metric 1). Thus, the slicing technique can improve the efficiency of the invariant checking (refer to

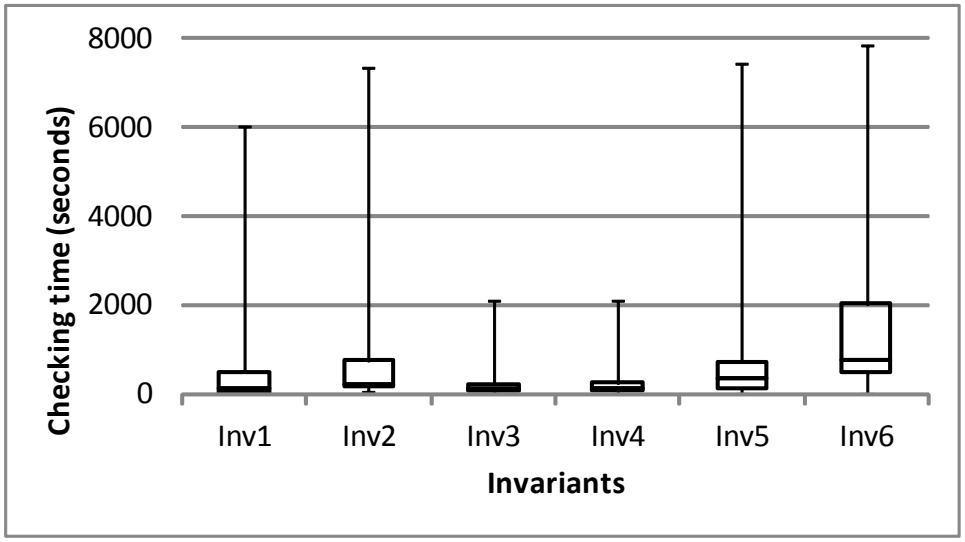


Figure 8 – Box plot for the measurement of Checking Time for Unsliced Metamodel and Models (*CTUM*)

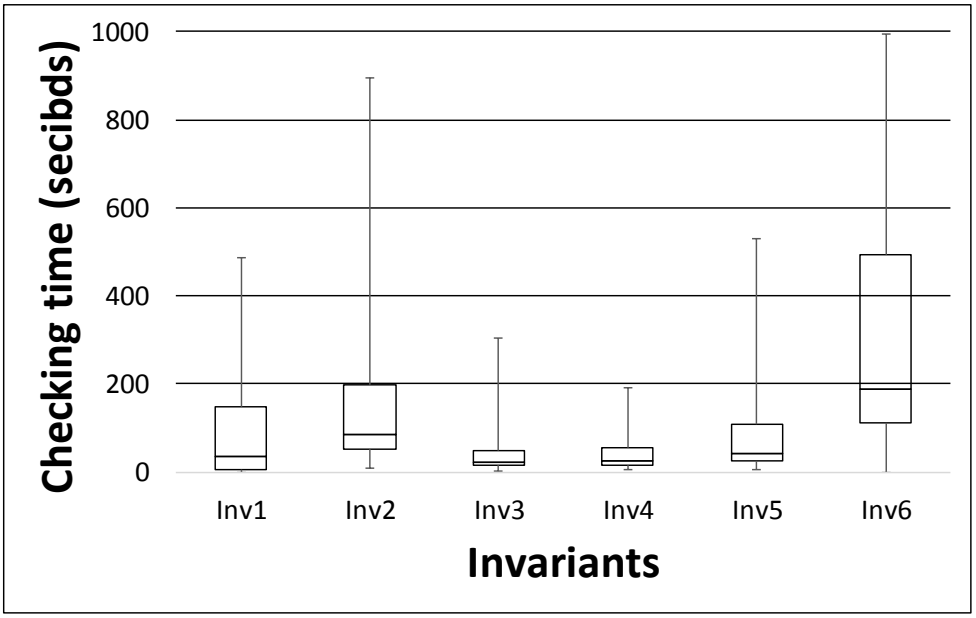


Figure 9 – Box plot for the measurement of Checking Time for Sliced Metamodel and Models (*CTSM*)

RQ1). Figure 11 also shows how significantly the slicing technique improves the checking efficiency. In the worst case scenario, the *CTS* is close to 1.5 (see the minimum *CTS* for *Inv1*), while in the best case scenario, the *CTS* is close to 36.0 (see the maximum *CTS* for *Inv4*). Since the first quartile of *CTSs* for each invariant is above 2.0, the slicing technique can significantly improve the checking efficiency for three fourths of the models used in the evaluation (i.e., 55

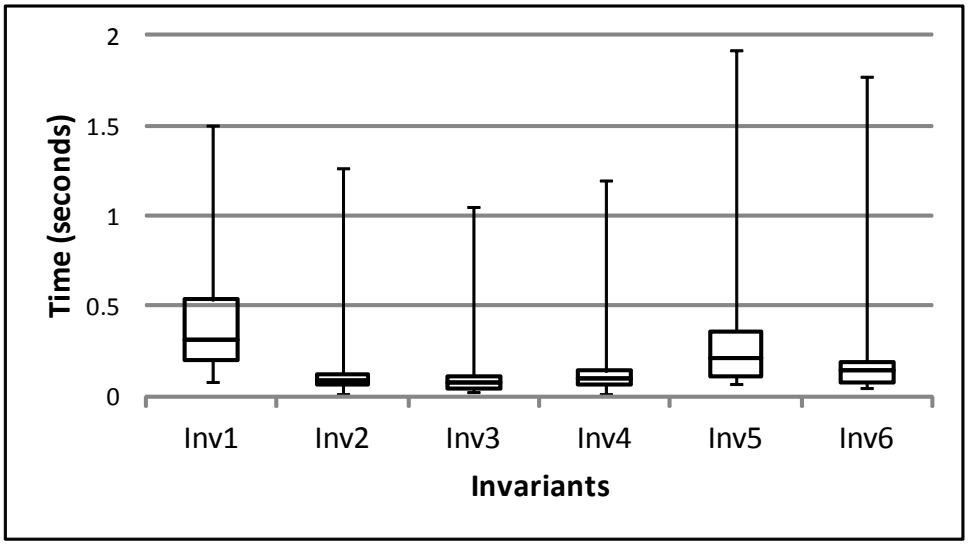


Figure 10 – Box plot for the measurement of *ST*

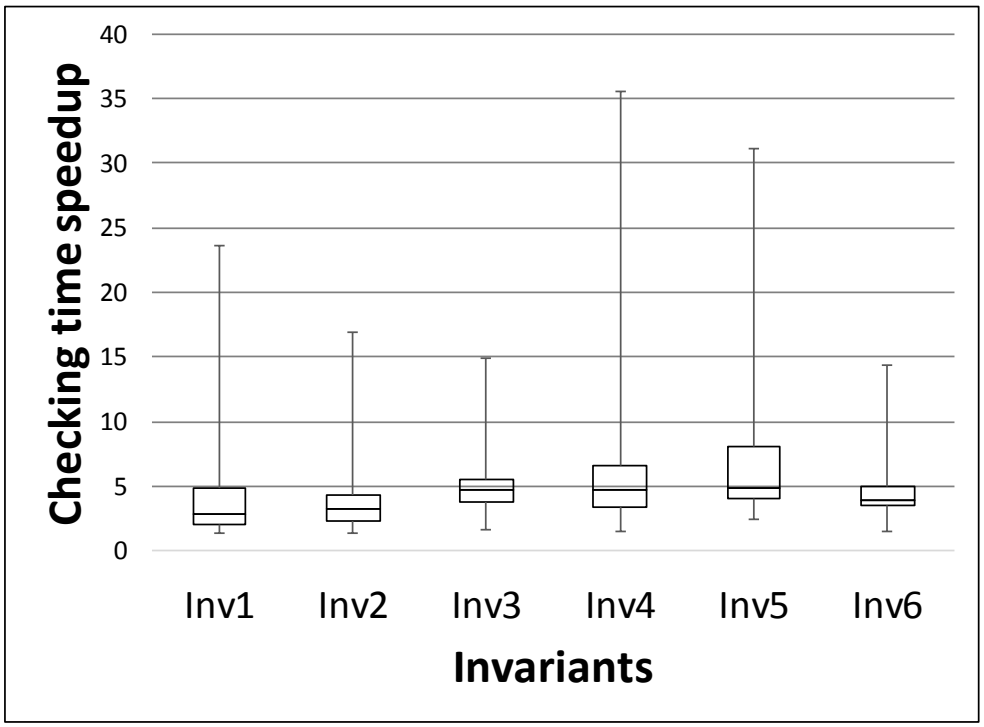


Figure 11 – Box plot for the measurement of Checking Time Speedup (CTS)

models). In summary, the slicing technique can reduce the time used for the invariant checking.

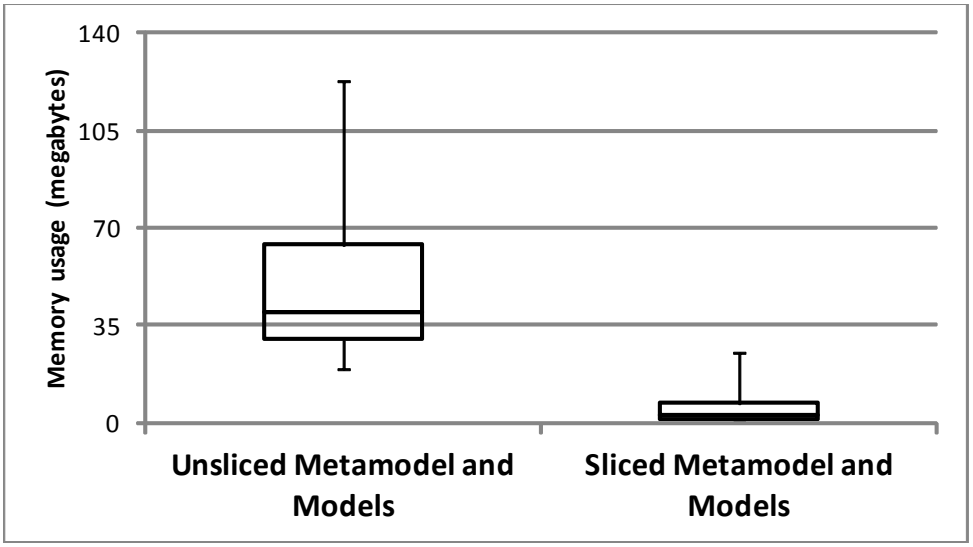


Figure 12 – Box plot for the measurement of the memory usage in the context of unsliced and sliced models w.r.t. *Inv6*

Note that the proposed slicing technique can also reduce the memory used for the invariant checking. For example, Figure 12 shows the distribution of the memory usage in the context of unsliced and sliced models w.r.t. *Inv6*. The memory used for checking unsliced models varies from 19 to 123 megabytes, while the memory used for checking sliced models varies from one to 25 megabytes. The median memory usage for unsliced models is 40 megabytes, while the median memory usage for sliced models is three megabytes.

In this section we also show that the invariant checking approach described in the paper can offer similar performance gains on small manually built models (e.g. hundreds of elements). Figure 13 shows the distribution of the checking time speedup for each invariant in the context of the manually built models. All the *CTSs* are calculated using Metric 1. The *CTS*s for the five models vary from 1.1 (minimum CTS) to 11.1 (maximum CTS). Since the *CTS* for each pair of invariant and model is above 1.0, the value of $CTSM + ST$ must be smaller than *CTUM* (refer to Metric 1). Thus in summary, the slicing technique can improve the efficiency of the invariant checking for small models (refer to *RQ1*).

5.3.2 Correctness of the Slicing Technique

The correctness of the slicing technique described in the paper depends on the invariant checking results. In other words, the proposed slicing technique is *correct* only if the checking results for the unsliced metamodel and model are the same as the checking results for the sliced metamodel and model. Thus to check whether the slicing technique is correct, we need to understand how the invariant checking works, and to measure the checking results for the unsliced and sliced models.

Typically the invariant checking proceeds in two steps. First, given an invariant and a model, it checks each object in the model and matches an object with the invariant if the object is an instance of a class in the context of which the invariant is defined. Second, it checks the matched object for the invariant and identifies the matched object as a valid object w.r.t. the invariant if the matched object satisfies the invariant. Therefore, given an invariant and a model, the invariant checking would return a result, showing the set of the matched objects and

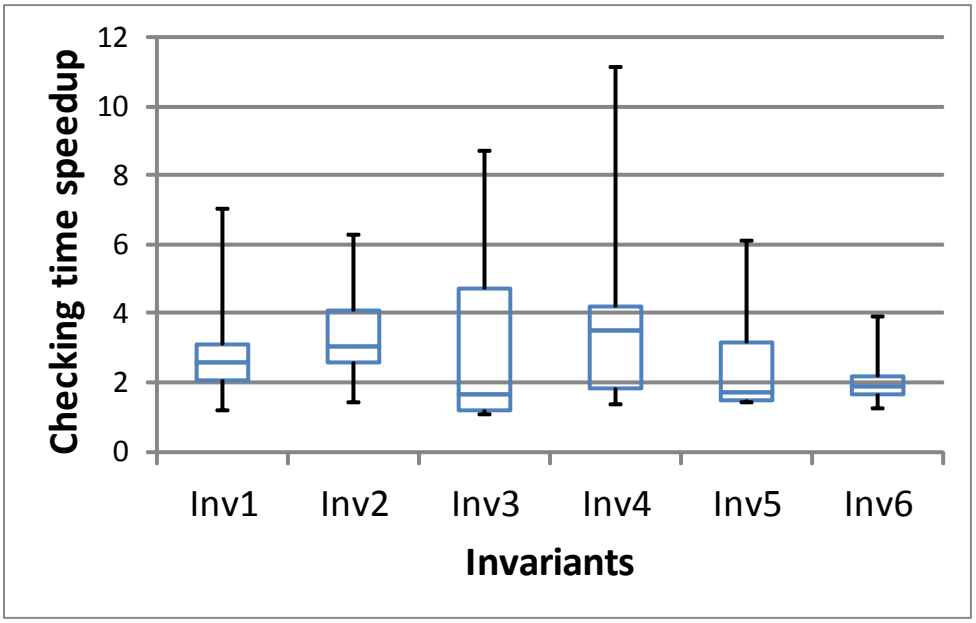


Figure 13 – Box plot for the measurement of Checking Time Speedup (CTS)

valid objects for the invariant. Note that the valid object set is always included in the matched object set (see Metric 2).

$$MatchedObjects \supseteq ValidObjects \tag{2}$$

Thus given an invariant, if the checking results for the unsliced model are the same as the checking results for the sliced model, the unsliced model and sliced model must have the same set of the matched and valid objects for the invariant. Therefore, we can use the conditions below to check the correctness of the proposed slicing technique. Given an invariant, the slicing technique is correct w.r.t. the checking results only if:

1. its matched objects in the unsliced model are the same as the matched objects in the sliced model;

2. its valid objects in the unsliced model are the same as the valid objects in the sliced model.

Thus to check the correctness of the proposed slicing technique, we need to identify both matched and valid objects for each invariant in unsliced and sliced models respectively. We used ID injection to accurately check whether an object in the unsliced model corresponds to an object in the sliced model. For example, at the metamodel level, we added an ID attribute into the top level class of the Java metamodel, and we kept the top level class with its ID attribute in the sliced Java metamodel. At the model level, we generate a unique ID for each object in the model. The evaluation results showed both unsliced and sliced models have the same set of matched and valid objects for each invariant. In summary, the emprical evaluation conforms that the slicing technique preserves the invariant checking results (refer to *RQ2*).

Other than the empirical confirmation, we can also show that the slicing technique is correct w.r.t. the checking results. Suppose that given a model and an invariant, we denote

all the model elements (e.g., objects, links and slots) in the model as *Obj_max*. In addition, the model elements that are needed for the invariant checking are denoted as *Elmt_min*. The relation bewteen *Elmt_max* and *Elmt_min* is given below:

$$Elmt\_max \supseteq Elmt\_min \tag{3}$$

In other words, this relation indicates that the checking results in the context of *Elmt_max* should be the same as the checking results in the context of *Elmt_min*. Our slicing technique can generate a sliced model that contains enough model elements for the invariant checking. We denote the model elements in the sliced model as *Elmt_slicing*. *Elmt_slicing* is a superset of *Elmt_min* and is a subset of *Elmt_max*. The relation among these three sets is given below:

$$Elmt\_max \supseteq Elmt\_slicing \supseteq Elmt\_min \tag{4}$$

Since the checking results for the *Elmt_max* and the *Elmt_min* are the same, the checking results for the *Elmt_max* and the *Elmt_slicing* should be the same. In summary, checking an invariant in the sliced model is equivalent to checking it in the unsliced model, and the proposed sliced models are sufficient to the invariant checking.

5.4 Threats to Validity

Construction threats lie in the way we defined the formulas used in the evaluation. The choices of formula and statistical analysis may have impact on evaluation results and conclusions. For example, Metric 1 does not take the model loading time into consideration. The reason we made this choice is because the model loading time is relatively small compared with the checking time (e.g., seconds v.s. minutes/hours). Also, models may be already loaded when performing the invariant checking.

The validity of the evaluation results may also be affected by calculations performed by the evaluation framework. To mitigate this threat, we calculated the *CTS* ten times for each pair of invariant and model, and used its average value. We also used different sizes of Java models (see Figure 7) in the evaluation to ensure the results are reliable. In addition, the use of the MoDisco tool [BCDM14] could be a threat to the validity of the evaluation because the correctness of the reverse engineering algorithm used in the tool has not been verified, and thus it may introduce errors to the models that are generated from the Java programs.

Another threat to validity we identified is the mono-operation threat, that is, only one metamodel was used in the evaluation. To mitigate this threat, we selected invariants that use different structural parts of the metamodel.

6 Discussion

The invariant checking approach described in Section 4 can have different variations based on the number of input invariants (Section 6.1), the way of footprinting (Section 6.2), the way of slicing the model (Section 6.3), and the checking context (Section 6.4).

In the remainder of this section we look into these variations.

6.1 Single v.s. Multiple Invariants

The invariant checking approach described in the paper can be used for a single invariant or multiple invariants. The illustration example given in Section 4 shows how our approach handles single invariant input. If the input includes multiple invariants (e.g., *N* invariants),

our approach generates a sliced metamodel and model for each invariant, and performs the checking *N* times. Thus the problem of checking multiple invariants can be reduced to the problem of checking single invariant.

It is possible to perform footprinting and slicing only once (e.g., generate one sliced metamodel and one sliced model for multiple invariants), and use the generated sliced metamodel and model in each invariant checking. In this case, the checking time would increase since the sliced model for multiple invariants could be larger than the sliced model for a single invariant (the model usage increases). However, since the time saved for footprinting and slicing (i.e., seconds) is much smaller than the time increased for checking (e.g., minutes), it is not worth checking one sliced model against multiple invariants.

It is also possible to (1) produce a single invariant from multiple invariants using conjunction, and (2) checking the conjunctive invariant against the model. However, it may cause two problems. First, the accuracy of the checking results would be decreased. For example, if a model violates the conjuncted invariant it is not clear which conjunct has been violated by the model. Second, checking the conjuncted invariant would be more inefficient compared to checking multiple invariants one by one as the size of the metamodel and model will be significantly larger. For example, one would use the operation *allInstances()* to merge two invariants with different contexts (e.g., see the *MinCoachSize* and *MinPassengers* invariants in Table 1) into one conjuncted invariant (see the *ConjunctedInv* invariant below).

Context *RegularTrip* **inv ConjunctedInv:**
self.passengers→size() $\geq$ 6 and Coach.allInstances()→forAll(c|c.noOfSeats $\geq$ 10)

If the checking complexity of *MinCoachSize* and *MinPassengers* is O(N) and O(M), the checking complexity of *ConjunctedInv* would be O(N * M) while the total checking complexity of *MinCoachSize* and *MinPassengers* would be O(N + M). Indeed, every time the tool checks an instance of *RegularTrip*, it checks all the instances of *Coach*.

6.2 Static v.s. Dynamic Footprinting

The way footprints are computed can be categorized into two types: static footprinting and dynamic footprinting [JGB11]. Static footprinting uses only the information from the invariant to guide the footprinting process. The model footprinting described in the paper is an example of static footprinting [JGB11]. Dynamic footprinting uses the model to identify an invariant that is applicable to the models (referred to as checking relevant invariant). The intuition behind this is based on the following observation: if each element in a model is not an instance of any metamodel element that is referred by an invariant, there is no need to check the model against the invariant because the model will not violate the invariant.

The dynamic footprinting is helpful for achieving higher *CTS* (see Metric 1). For example, suppose that an invariant checking is irrelevant with respect to a model. There is thus no need to check the model against the invariant (i.e., *CTSM* is 0). In addition, the time used to analyze models is quite small (e.g., seconds). Therefore, *CTS* would be significantly improved in this case.

6.3 Aggressive v.s. Conservative Metamodel Slicing

In aggressive slicing, the sliced metamodel only contains the elements in the footprint (i.e., slicing criterion) and the subclasses of the classes in the footprint. In conservative slicing (also called pruning [SMBJ09]), the sliced metamodel also contains the classes that are types of the

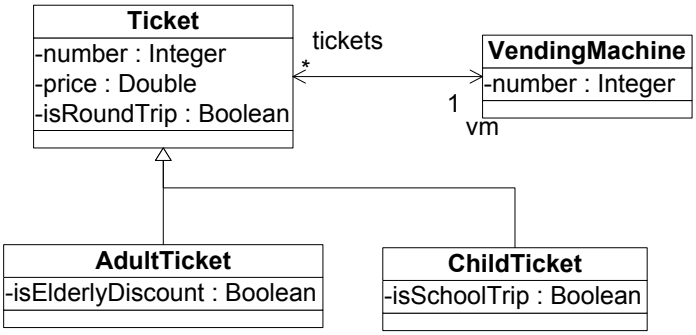


Figure 14 – A modified version of the metamodel in Figure 1 (the multiplicity of the *vm* reference has been changed, and most classes in Figure 1 are omitted)

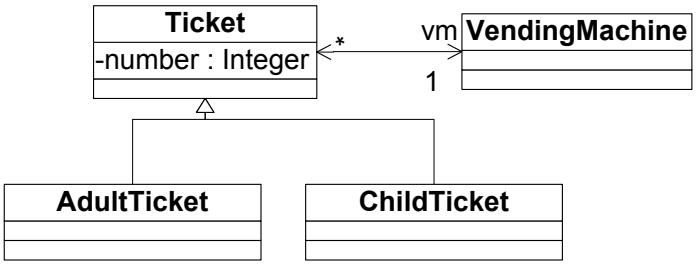


Figure 15 – A sliced metamodel generated from the *UniqueTicketNumber* invariant and the metamodel in Figure 14 using the conservative slicing

mandatory references (i.e., the lower bound of the multiplicity of the reference must be equal to or larger than 1) from the classes in the footprint.

For example, Figure 14 shows a modified version of the metamodel in Figure 1. Note that the multiplicity of the *vm* reference in Figure 1 is 0..1, while that in Figure 14 is 1. The *vm* reference in Figure 14 is an example of the mandatory reference.

Given the *UniqueTicketNumber* invariant and the metamodel in Figure 14, the metamodel (see Figure 3 (left)) generated using the aggressive slicing contains only the *Ticket* class and its subclasses, while the metamodel (see Figure 15) generated using the conservative slicing contains the mandatory reference (i.e., *vm*) and its type class (i.e., *VendingMachine*).

The invariant checking approach described in the paper uses the aggressive slicing. This is mainly because the aggressive slicing could produce smaller metamodel and model and thus reduce the checking time. However, the aggressive slicing could be a threat to the correctness of the invariant checking approach described in the paper if the input model is not a valid instance of the input metamodel.

For example, Figure 16 shows a model that is not a valid instance of the metamodel in Figure 14. Given the model in Figure 16 and the metamodel in Figure 14, the checking results should return false since a ticket needs to be associated with a vending machine. However, if we use the metamodel in Figure 3 (left) (i.e., the result of the aggressive slicing for the *UniqueTicketNumber* invariant) to slice the model in Figure 16, and check the sliced model against the *UniqueTicketNumber* invariant in the context of the metamodel in Figure 3, the checking results would return true, which contradicts the checking results without using the slicing technique. If we use the metamodel in Figure 15 to slice the model in Figure 16, and

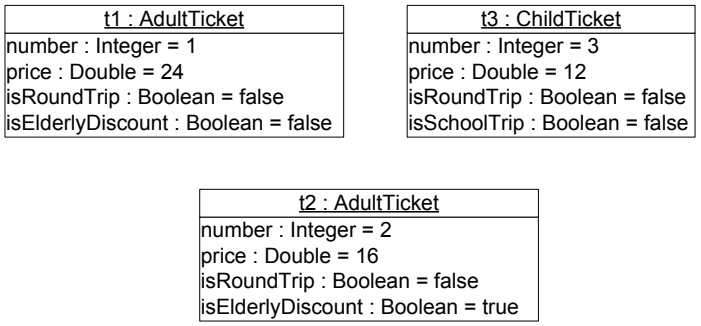


Figure 16 – A model that is not a valid instance of the metamodel in Figure 14

check the sliced model, the checking results would return false, which is consistent with the checking results without using the slicing technique. Therefore, to use the aggressive slicing, we need to add a precondition to our invariant checking approach, that is, the input model must be a valid instance of the input metamodel (see the precondition in the Introduction Section).

6.4 Checking in the Context of the Entire Metamodel v.s. the Sliced Metamodel

In the invariant checking approach described in section 4, the sliced model is checked in the context of the sliced metamodel. The reasons are twofold. First, the sliced metamodel is smaller than the entire metamodel, and the invariant checking in the context of the sliced metamodel would take less time. Second, the aggressive slicing could produce models that are not valid instances of the entire metamodel, and the checking results for such models in the context of the entire metamodel always return false.

For example, given the *UniqueTicketNumber* invariant, the model in Figure 2 and the metamodel in Figure 14, the aggressive slicing would produce a sliced metamodel in Figure 3 (left) and a sliced model in Figure 16. When the sliced model is checked in the context of sliced metamodel, the checking result would be true (every ticket has a unique number), which is consistent with the checking result without using the slicing technique. However, if the sliced model is checked in the context of the entire metamodel in Figure 14, the checking result would be false, because a ticket needs to be associated with a vending machine, and the sliced model is not a valid instance of the metamodel in Figure 14. Therefore, to check the sliced model in the context of the entire metamodel, we need to use the conservative slicing in the invariant checking approach.

7 Related Work

In this section, we present the related work that use the slicing technique to improve the performance of a variety of modeling tasks.

Uzuncaova et al. [UK07] proposed a slicing technique for the Alloy model [Jac02]. The technique can reduce the time used to solve the satisfiablity problem, that is, the problem that involves checking whether there is an instance for a given Alloy model. Unlike the slicing technique described in the paper, their approach can be only used for the declarative language like Alloy, and cannot be used for the invariant checking.

Shaikh et al. [SCWM10][SWM11] used a slicing technique to improve the scalability of an analysis that involves checking if a metamodel has a valid instance that satisfies the invariants defined in the metamodel. The slicing technique reduces the analysis problem of checking a large metamodel with OCL invariants into smaller subproblems, where each subproblem involves checking a metamodel fragment with a subset of OCL invariants. Unlike the slicing technique described in the paper, their technique focuses only on metamodel with invariants, and cannot be used to slice instances of a metamodel (i.e. models).

Egyed et al. [Egy06] described an instance-based consistency checking approach for design models. The approach uses the changes in a model to determine whether a given consistency rule is needed to be reevaluated for the model. Compared with the approaches that evaluate the model against all the consistency rules, their approach can reduce the number of rules being evaluated, and thus significantly improve the consistency checking. Their way of eliminating the checking irrelevant rules is similar to the dynamic footprinting version of our approach. The only difference is that they use the changes performed in the model to determine the checking relevant rules, and we use the entire model to determine the checking relevant invariants.

Garcia [Gar08] described an efficient integrity checking approach for models in software repositories. Cabot et al. [CT09] incrementally checked the integrity of the UML and OCL conceptual schemas (i.e., metamodels). However, unlike the approach described in the paper that builds upon the slicing technique, their approaches used the incremental evaluation to improve the efficiency of invariant checking.

8 Conclusion

In this section we summarize the contribution and limitations of our work and points to the future directions.

8.1 Contribution and Limitation

We introduced a slicing technique to improve the efficiency of model analysis that involves checking if an instance of a metamodel satisfies the invariants defined in the metamodel. The technique uses the invariant being checked in order to produce a sliced metamodel and model from the input metamodel and model. The sliced metamodel and model so produced is much smaller in size which allows more efficient checking (both in terms of memory and timing) by using existing invariant checking tools. Our experiments revealed that the proposed slicing technique can significantly reduce the time to perform the invariant checking, achieving checking speedup ranging from 1.5 to 36.0.

Our slicing technique is limited in its ability to produce smaller footprint from a larger metamodel if the invariant requires all the metamodel elements to be present when analyzed – this happens when there is a very tight coupling across all metamodel elements. However, it has been confirmed by [CCB12] that a substantial number of invariants only reference part of the metamodel in which they are defined.

8.2 Research Agenda

The slicing technique described in the paper is used to improve the efficiency of invariant checking techniques. However, invariant checking is not the only usage scenario for the slicing technique. The slicing technique can be further improved and used for other modeling tasks that involve both metamodels and models.

Another future direction of this work could be slicing invariants using the information found in models. For example, a complex invariant may reference a substantial number of metamodel elements, while a model may only reference a small subset of metamodel elements. In this case, checking the model against the invariant would not require the entire invariant to be analyzed. Thus, the slicing technique would use the information in the model to reduce the complex invariant into smaller subinvariants, where only a subset of the subinvariants are needed to analyze the model.

References

[ABC+11] K. Androutsopoulos, D. Binkley, D. Clark, N. Gold, M. Harman, K. Lano, and Z. Li. Model projection: Simplifying models in response to restricting the environment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 291–300, New York, NY, USA, 2011. ACM. doi:10.1145/1985793.1985834.

[BCBB11] A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux. Modeling Model Slicers. In *ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems*, volume 6981, pages 62–76. Springer Berlin / Heidelberg, 2011. doi:10.1007/978-3-642-24485-8_6.

[BCBB12] A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux. Kompren: Modeling and Generating Model Slicers. *Software and Systems Modeling*, pages 1–17, October 2012. doi:10.1007/s10270-012-0300-x.

[BCDM14] H. Brunelière, J. Cabot, G. Dupé, and F. Madiot. Modisco: A model driven reverse engineering framework. *Information and Software Technology*, 56(8):1012 – 1032, 2014. doi:10.1016/j.infsof.2014.04.007.

[CCB12] J. Cadavid, B. Combemale, and B. Baudry. Ten years of Meta-Object Facility: an analysis of metamodeling practices. *Technical Report by the Triskell Team at INRIA/IRISA*, (RR-7882):1–25, 2012. URL: https://hal.inria.fr/hal-00670652.

[CT09] J. Cabot and E. Teniente. Incremental Integrity Checking of UML/OCL Conceptual Schemas. *Journal of Systems and Software*, 82(9):1459 – 1478, 2009. doi:http://dx.doi.org/10.1016/j.jss.2009.03.009.

[EFLR99] A. Evans, R. France, K. Lano, and B. Rumpe. The uml as a formal modeling notation. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language. «UML»'98: Beyond the Notation*, volume 1618 of *Lecture Notes in Computer Science*, pages 336–348. Springer Berlin Heidelberg, 1999. doi:10.1007/978-3-540-48480-6_26.

[Egy06] A. Egyed. Instant Consistency Checking for the UML. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 381–390, 2006. doi:10.1145/1134285.1134339.

[EW04] R. Eshuis and R. Wieringa. Tool Support for Verifying UML Activity Diagrams. *IEEE Trans. Softw. Eng.*, 30(7):437–447, 2004. doi:10.1109/TSE.2004.33.

[Gar08] M. Garcia. Efficient Integrity Checking for Essential MOF+ OCL in Software Repositories. *Journal of Object Technology*, 7(6):101–119, 2008. URL: http://www.jot.fm/issues/issue_2008_07/article3.pdf.

[GBR07] M. Gogolla, F. Büttner, and M. Richters. USE: A UML-based Specification Environment for Validating UML and OCL. *Sci. Comput. Program.*, 69(1-3):27–34, 2007. doi:10.1016/j.scico.2007.01.013.

[GL91] K.B. Gallagher and J.R. Lyle. Using program slicing in software maintenance. *Software Engineering, IEEE Transactions on*, 17(8):751–761, 1991. doi:10.1109/32.83912.

[Jac02] D. Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002. doi:10.1145/505145.505149.

[JCB+13] J. Jézéquel, B. Combemale, O. Barais, M. Monperrus, and F. Fouquet. Mashup of metalanguages and its implementation in the kermeta language workbench. *Software and Systems Modeling*, pages 1–16, 2013. URL: http://dx.doi.org/10.1007/s10270-013-0354-4, doi:10.1007/s10270-013-0354-4.

[JGB11] C. Jeanneret, M. Glinz, and B. Baudry. Estimating footprints of model operations. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 601–610. IEEE, 2011. doi:10.1145/1985793.1985875.

[KMS05] H. Kagdi, J.I. Maletic, and A. Sutton. Context-Free Slicing of UML Class Models. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 635–638. IEEE Computer Society, 2005. doi:10.1109/ICSM.2005.34.

[KSTV03] B. Korel, I. Singh, L. Tahat, and B. Vaysburg. Slicing of state-based models. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 34–43, 2003. doi:10.1109/ICSM.2003.1235404.

[LKR10] K. Lano and S. Kolahdouz-Rahimi. Slicing of UML Models Using Model Transformations. In *Model Driven Engineering Languages and Systems*, volume 6395 of *Lecture Notes in Computer Science*, pages 228–242. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-16129-2_17.

[LKR11] K. Lano and S. Kolahdouz-Rahimi. Slicing Techniques for UML Models. *Journal of Object Technology*, 10:11:1–49, 2011. doi:10.5381/jot.2011.10.1.a11.

[Omg08] QVT Omg. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. *Final Adopted Specification (November 2005)*, 2008. URL: http://www.omg.org/spec/QVT/.

[SBMP09] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2009. URL: http://dl.acm.org/citation.cfm?id=1197540.

[SCWM10] A. Shaikh, R. Clarisó, U.K. Wiil, and N. Memon. Verification-driven slicing of UML/OCL models. In *Proceedings of the IEEE/ACM international conference on Automated Software Engineering*, ASE '10, pages 185–194. ACM, 2010. doi:10.1145/1858996.1859038.

[SFR11] W. Sun, R. France, and I. Ray. Rigorous Analysis of UML Access Control Policy Models. In *Policies for Distributed Systems and Networks (POLICY), 2011 IEEE International Symposium on*, pages 9–16, 2011. doi:10.1109/POLICY.2011.30.

[SFR13] W. Sun, R. France, and I. Ray. Contract-Aware Slicing of UML Class Models. In *Model-Driven Engineering Languages and Systems*, volume 8107 of *Lecture*

Notes in Computer Science, pages 724–739. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-41533-3_44.

[SMBJ09] S. Sen, N. Moha, B. Baudry, and J. Jézéquel. Meta-model Pruning. In *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 32–46. Springer Berlin Heidelberg, 2009. doi:10.1007/978-3-642-04425-0_4.

[Spe07a] O.M.G.A. Specification. Object Constraint Language (OCL), 2007. URL: http://www.omg.org/spec/OCL/.

[Spe07b] O.M.G.A. Specification. XML Metadata Interchange (XMI), 2007. URL: http://www.omg.org/spec/XMI/.

[SWM11] A. Shaikh, U.K. Wiil, and N. Memon. Evaluation of Tools and Slicing Techniques for Efficient Verification of UML/OCL Class Diagrams. *Adv. Soft. Eng.*, 2011:5:1–5:18, 2011. doi:10.1155/2011/370198.

[Tea05] Eclipse OCL Project Team. Eclipse OCL Project. Eclipse Community, 2005. URL: http://projects.eclipse.org/projects/modeling.mdt.ocl.

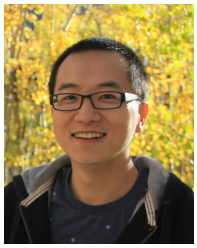
[Tea13a] AtlanMod Team. Metamodel Zoos. AtlanMod Team, 2013. URL: http://www.emn.fr/z-info/atlanmod/index.php/Zoos.

[Tea13b] ReMoDD Team. Repository for Model Driven Development (ReMoDD) Overview. Repository for Model-Driven Development (ReMoDD), 2013. URL: http://www.cs.colostate.edu/remodd/v1/content/repository-model-driven-development-remodd-overview.

[UK07] E Uzuncaova and S. Khurshid. Kato: A program slicing tool for declarative specifications. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 767–770, May 2007. doi:10.1109/ICSE.2007.47.

[Wei81] M. Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449. IEEE Press, 1981. URL: http://dl.acm.org/citation.cfm?id=800078.802557.

About the authors



Wuliang Sun received his Ph.D. in Computer Science from Colorado State University, advised by Robert B. France and Indrakshi Ray. Before that, he received an M.S. in Computer Science from Baylor University, advised by Eunjee Song, and a B.S. in Computer Science from Sichuan University. He has worked as a visiting researcher at INRIA/IRISA. His research interests are in Software Engineering, focusing on software modeling, model analysis, and model slicing. Contact him at sunwl@cs.colostate.edu

Benoit Combemale is an Associate Professor at University of Rennes 1, France. He is also on secondment at INRIA as research computer scientist (2013-2016). He earned a PhD in Computer Science (2005-2008) from University of Toulouse awarded by the prize Leopold Escande 2008. He is working within the research team DiverSE (formerly Triskell) at INRIA/IRISA, Rennes. His research interests include model-driven engineering (MDE), and software language engineering (SLE). Contact him at benoit.combemale@irisa.fr.

Robert B. France is a Professor in the Department of Computer Science at Colorado State University. He received his B.Sc. in Natural Sciences with First Class Honors from the University of the West Indies in Trinidad and Tobago. He attended Massey University in New Zealand under a Commonwealth Scholarship, where he graduated with a Ph.D. in Computer Science. His research on model-driven software development focuses on providing software developers with mathematically-based software modeling languages and supporting analysis tools that they can use to specify and analyze critical software properties (e.g., behavioral and security properties). He is a founding editor-in-chief of the Springer journal on Software and Systems Modeling, and a founding steering committee member of the international conference series on Model Driven Engineering Languages and Systems (MODELS). Contact him at france@cs.colostate.edu.

Arnaud Blouin is an Associate professor at the National Institutes of Applied Sciences (INSA), Rennes, France. He received a PhD degree in Computer Science from the University of Angers, France, in 2009. After a post-doc at INRIA, he joined INSA in 2011. His research interests within the DiverSE research group include model-driven engineering, interactive system engineering, software testing, and software comprehension. Contact him at arnaud.blouin@irisa.fr.

Benoit Baudry is a research scientist at INRIA. He received a PhD degree in Computer Science from the University of Rennes, France, in 2003. He first worked at CEA (French center for atomic energy) before joining INRIA in 2004. His research interests include software testing and verification, model-driven engineering, and requirements analysis. He leads the DiverSE research group (EPI), which investigates model-driven engineering and software product lines from requirements to runtime. Contact him at benoit.baudry@inria.fr.

Indrakshi Ray is a Professor in the Computer Science Department at Colorado State University. She has also been a visiting faculty at Air Force Research Laboratory and at INRIA, Rocquencourt, France. She obtained her Ph.D. from George Mason University under the joint supervision of Professor Sushil Jajodia and Professor Paul Ammann. Her research interests include security and privacy, database systems, e-commerce and formal methods in software engineering. She has published over a hundred technical papers in refereed journals and conference proceedings. She is on the editorial board of Computer Standards and Interfaces. She was the Program Chair of ACM SACMAT 2006, Program Co-Chair for CSS 2013, ICISS 2013, IFIP DBSec 2003, and General Chair of SACMAT 2008. Contact her at iray@cs.colostate.edu.

Acknowledgments This work was supported by the National Science Foundation grant (CCF-1018711), the ANR INS Project GEMOC (ANR-12-INSE-0011), and the CNRS PICS Project MBSAR.