DISSERTATION

USING SLICING TECHNIQUES TO SUPPORT SCALABLE RIGOROUS ANALYSIS OF CLASS MODELS

Submitted by

Wuliang Sun

Department of Computer Science

In partial fulfillment of the requirements

for the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2014

COLORADO STATE UNIVERSITY

June, 2014

WE HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER OUR SUPERVISION BY WULIANG SUN ENTITLED USING SLICING TECHNIQUES TO SUPPORT SCALABLE RIGOROUS ANALYSIS OF CLASS MODELS BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

Committee on Graduate Work

_____
Committee Member

_____
Committee Member

_____
Adviser

_____
Co-Adviser

_____
Department Head

ABSTRACT OF DISSERTATION

USING SLICING TECHNIQUES TO SUPPORT SCALABLE RIGOROUS ANALYSIS OF CLASS MODELS

Slicing is a reduction technique that has been applied to class models to support model comprehension, analysis, and other modeling activities. In particular, slicing techniques can be used to produce class model fragments that include only those elements needed to analyze semantic properties of interest. However, many of the existing class model slicing techniques do not take constraints (invariants and operation contracts) expressed in auxiliary constraint languages into consideration when producing model slices. Their applicability is thus limited to situations in which the determination of slices does not require information found in constraints.

In this dissertation we describe our work on class model slicing techniques that take into consideration constraints expressed in the Object Constraint Language (OCL). The slicing techniques described in the dissertation can be used to produce model fragments that each consists of only the model elements needed to analyze specified properties. The slicing techniques are intended to enhance the scalability of class model analysis that involves (1) checking conformance between an object configuration and a class model with specified invariants and (2) analyzing sequences of operation invocations to uncover invariant violations. The slicing techniques are used to produce model fragments that can be analyzed separately. An evaluation we performed provides evidence

that the proposed slicing techniques can significantly reduce the time to perform the analysis.

Wuliang Sun
Department of Computer Science
Colorado State University
Fort Collins, CO 80523
Summer 2014

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

1.1 Problem

In the model-driven development (MDD) area, models are the major artifacts that drive the software development process. Rigorous analysis of models can enhance the ability of developers to understand the system under development and to identify potentially costly design problems earlier. Class models expressed using the Unified Modeling Language (UML) [13] are among the most popular models used in practice, and given their pivotal roles, a number of tool-supported rigorous analysis techniques have been proposed (e.g., see [15][37][19][18][12][21][31]) for UML class models. However, given that the complexity of software systems is increasing, one can expect that class models used to represent these complex systems will also grow significantly in size. The scalability of current class model analysis tools will become an issue in these situations, and thus there is a need for techniques that support scalable rigorous analysis of UML class models.

Slicing techniques [51] 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 [14][51]), and models (e.g., see [3][6][11][22][24]). In the MDD area, model slicing techniques have been used to support a variety of modeling tasks, including model comprehension [3][6][24], analysis

[20][27][28], and verification [11][40][43].

In model slicing techniques *slicing criteria* are 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 (i.e., elements satisfy a *slicing criterion*) and the rest of a 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 and their dependent model elements, is extracted from the model.

Rigorous analysis of invariants and operation contracts expressed in the Object Constraint Language (OCL) [44] can be expensive when the class models are large. Model slicing techniques can be used in these situations to reduce large models to just those fragments that can be analyzed separately. This reduction can help reduce the cost of analysis. However, many of the existing class model slicing techniques do not take constraints expressed in auxiliary constraint languages into consideration when producing model slices. Their applicability is thus limited to situations in which the determination of slices does not require information found in constraints. This limits the utility of class model slicing techniques in (1) situations where invariants are needed to specify additional properties that cannot be expressed directly in a class model (e.g., acyclicity property), and (2) model-based software development approaches that are contract based (e.g., design by contract [30]).

1.2 Solution

In this dissertation we describe class model slicing techniques that take into consideration invariants and operation contracts expressed in the OCL. The techniques are used to produce model fragments, each of which consists of only the model elements needed to analyze specified properties. We have developed the slicing techniques to enhance the

scalability of two useful class model analysis techniques, (1) a technique for checking conformance between an object configuration and a class model with specified invariants and (2) a technique for analyzing sequences of operation invocations to uncover invariant violations.

As stated above we propose a slicing technique to improve the efficiency of a model analysis technique that checks conformance between an object configuration and a class model with invariants. Such analysis plays an important role in MDD in that it can improve developers' understanding of complex systems and uncover structural errors in design class models during the early stages of software development. General-purpose rigorous analysis tools can be used to check conformance, however, they are likely to perform the analysis using the entire class model and full object configurations. Their scalability thus becomes an issue when the sizes of the class models and object configurations are large. In this dissertation we describe how domain knowledge in the form of model slicing criteria can be used to improve the scalability of these tools. Here, domain knowledge refers to class model, object configuration, or constraint elements used to formulate slicing criteria for the class model and object configuration involved in an analysis. Specifically, the approach encompasses slicing variations ranging from slicing that utilizes only elements in a single invariant and the class model to produce a sliced object configuration and class model (static, single invariant slicing), to slicing that utilizes elements in both the class model and the object configuration, in addition to information in multiple invariants, to produce a sliced object configuration and class model (dynamic, multiple invariants slicing). This gives developers flexibility in determining the forms of domain knowledge used to break-up models and object configurations into fragments that can be more efficiently analyzed. The outputs of the slicing technique (i.e., the extracted model/object configuration fragment) can be fed into a general-purpose model analysis tool (e.g., Kermeta [12]) for conformance checking.

We also propose a slicing technique to improve the efficiency of a model analysis technique we developed to check that operation contracts do not allow invariant violations when sequences of conforming operations are invoked [46]. The slicing technique is used to reduce the problem of analyzing a large model with many OCL constraints (including OCL operation contracts) to smaller subproblems that involve analyzing a model fragment against a subset of invariants and operation contracts. Each model fragment can be analyzed independently of other fragments. Given a class model with OCL constraints, the slicing technique automatically generates slicing criteria consisting of a subset of invariants and operation contracts, and uses the criteria to extract model fragments. Each model fragment is obtained by identifying and analyzing relationships between model elements and the constraints included in a generated slicing criterion.

The analysis results will be preserved by the slicing techniques. This assertion is based on the observation that the technique produces the fragments by identifying the model elements that are directly referenced by OCL expressions and analyzing their dependencies with other model elements. The evaluation also provides some evidence (albeit, not formal) that the analysis results are preserved by the slicing techniques.

1.3 Contribution

The major contribution of the research is a model slicing platform that provides implementations of two model slicing techniques we have developed. Below is a list of contributions of the research:

1. A state-of-the-art survey on class model slicing techniques;

2. A rigorous technique that supports co-slicing of UML class models and object configurations;

3. A rigorous technique that supports slicing of UML class models including both

OCL invariants and operation contracts;

4. A platform that provides implementations of two proposed class model slicing techniques;

5. An evaluation framework for the proposed slicing techniques.

1.4 Dissertation Structure

The rest of the dissertation is organized as follows. Chapter 2 presents a systematic literature review on class model slicing techniques. Chapter 3 describes the class model analysis techniques whose scalability we aim to improve through slicing. Chapter 4 describes the class model slicing techniques. Chapter 5 presents a tool support for the proposed slicing techniques and their evaluation. Chapter 6 presents the results of an evaluation of the slicing techniques. Chapter 7 concludes the dissertation.

Chapter 2

Literature Review

In this chapter we present the result of a systematic literature review we conducted on class model slicing techniques. A systematic review is important for research activities since it summarizes existing techniques concerning a research interest and identifies further research directions [23]. The purpose of the review described in this chapter is to compare current class model slicing techniques and identify their limitations through a systematic evaluation. The review follows a carefully designed paper selection procedure, and identifies techniques in scientific journals and conferences from 2003 to 2013.

Kitchenham et al. [23] described a systematic procedure to identifying and analyzing available literature relevant to a specific research topic. Tao [56] applied this procedure to her doctoral research on deriving a UML analysis model from a use case model. The three-step approach we used is based on the work described in [23] and [56]. First, we determined the scope of the systematic review (Section 2.1), and identified the research questions to be answered by the review (Section 2.2). Second, we developed a search strategy (Section 2.3) for identifying relevant research papers. Third, we compared relevant publications on class model slicing techniques (Section 2.4) and performed an evaluation of the work they describe to answer the research questions identified earlier (Section 2.5).

The slicing techniques described in this dissertation aim to address open issues (Section 2.6) that are identified through the analysis of published work evaluated as part of our systematic literature review.

2.1 Review Scope

The research described in the dissertation aims to use slicing techniques to enhance the scalability of class model analysis. The scope of the systematic review can thus be restricted to research on slicing techniques.

The systematic review focuses on slicing techniques that can handle UML class models. Class models involved in the reviewed techniques must conform to the UML standard [13] and can have invariants and operation contracts expressed using the OCL [44]. Slicing techniques that take as inputs multiple models are also considered in the review if the inputs of the slicing techniques include class models.

2.2 Research Questions

The systematic review aims to answer the following research questions:

1. Purpose [PS]: What are the different techniques used for slicing class models? What are their purposes? It is important to understand the purpose of a slicing technique since the slicing purpose determines (1) the input of a slicing technique and (2) the form of a slicing result (i.e., a slice).

2. Class Model Type [CMT]: What are the different types of class models used in these slicing techniques? Four types of class models are supported by the existing slicing techniques: basic class model [Basic] (i.e., class model that does not include invariants or operation contracts), class model including invariants [Inv],

class model including operation contracts [Op], and class model including both invariants and operation contracts [InvOp].

3. Slicing Criterion [SC]: What are the different slicing criteria used by current class model slicing techniques? Any class model elements can be included in a slicing criterion depending on the slicing purpose.

4. Intermediate Model [IM]: Do any of the current techniques use intermediate models in the dependency analysis processes? For example, a slicing technique may use a dependency graph as an intermediate model to perform the dependency analysis. In this case, it may require extra effort for the slicing technique to transform a class model into a dependency graph.

5. Automation [AM]: Are current slicing techniques automated or automatable? A slicing technique is (1) automated if the slicing technique has a tool support, or (2) automatable if a slicing algorithm is presented in the paper.

2.3 Search Strategy

In this section we describe a two-step search strategy used to select papers that are relevant to the class model slicing topic. First, we identified a number of relevant papers using electronic and manual search (Section 2.3.1). Second, we refined the search result using a selection criteria described in Section 2.3.2.

2.3.1 Electronic and Manual Search

We performed electronic search within three electronic databases: IEEE Xplore, ACM Digital Library, and SpringerLink. The electronic search was done in three steps. First, we came up with a list of query strings that are related to class model slicing techniques. Second, each query string was used to search three electronic databases. Table 2.1

Table 2.1: Electronic Search Result (from 2003 to 2013)

Query Strings	IEEE Xplore	ACM Digital Library	SpringerLink
UML class model slicing	3	218	95
UML model slicing	18	265	109
UML class model decomposition	6	866	459
UML model decomposition	34	1088	596
Decompose UML class model	2	268	504
Decompose UML model	12	338	663

presents the number of papers that were found by the electronic search. The first column shows a list of query strings we used in the search. The remaining columns (from the second column to the fourth column) show the number of papers that were found in the IEEE Xplore, the ACM Digital Library, and the SpringerLink respectively. For example, given the "UML class model slicing" query string, three papers were return from the IEEE Xplore, 218 papers were returned from the ACM Digital Library, and 95 papers were returned from the SpringerLink.

As a complement to the electronic search, we performed a manual search in specific journals and conference proceedings. We manually searched all published papers from 2003 to 2013 in five potentially relevant, peer-reviewed journals: IEEE Transactions on Software Engineering (TSE), ACM Transactions on Software Engineering and Methodology (TOSEM), Requirements Engineering Journal (JRE), Journal of Systems and Software (JSS), and Software and Systems Modeling (SoSym). Table 2.2 presents the number of papers that were found from these journals.

We also manually searched all published papers from 2003 to 2013 in eight potentially related conference proceedings: ACM/IEEE International Conference on Software Engineering (ICSE), ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS), European Conference on Object-Oriented Programming (ECOOP), IEEE/ACM International Conference on Automated Software Engi-

Table 2.2: Manual Search Result from Journals (from 2003 to 2013)

Journals	TSE	TOSEM	JRE	JSS	SoSym
Paper Number	1	0	0	0	1

Table 2.3: Manual Search Result from Conferences (from 2003 to 2013)

Conferences	ICSE	FSE	MODELS	ECOOP	ASE	ICSM	RE	FASE
Paper Number	1	1	3	0	1	1	0	0

neering (ASE), IEEE International Conference on Software Maintenance (ICSM), IEEE International Requirements Engineering Conference (RE), and International Conference on Fundamental Approaches to Software Engineering (FASE). Table 2.3 presents the number of papers that were found from these conferences.

2.3.2 Exclusion/Inclusion Procedure

As a tremendous number of candidate papers resulted from the electronic and manual search, an exclusion procedure was used to refine the search result. The procedure takes as input a candidate paper, and accepts the paper if it is relevant to the research topic described in Section 2.1 (i.e., UML class model slicing):

1. If the paper describes a slicing technique, go to the next step; otherwise, reject the paper.

2. If the slicing technique is used for models, go to the next step; otherwise, reject the paper.

3. If the input of the slicing technique includes UML class models, accept the paper; otherwise, reject the paper.

The exclusion procedure was applied to the search result given in Section 2.3.1. We started the procedure by reading the title and abstract of a candidate paper. When a deci-

sion (i.e., accept or reject the paper) was not able to be made from the title and abstract of the paper, we further checked the paper's introduction and conclusion. In summary, 12 papers from the search result (i.e., [4][5][6][7] [20][22][26][27][39][40][41][42]) were identified as relevant to the research topic by the exclusion procedure.

An inclusion procedure was then applied to these 12 relevant papers. The purpose of the inclusion procedure is to identify additional papers that were not selected by the paper search strategy described in Section 2.3.1. The four-step inclusion procedure was performed on each of these 12 relevant papers. First, all the papers that are given in the reference list of a relevant paper were selected. Second, all the papers that reference the relevant paper were selected. The search engine, *scholar.google.com*, was used to identify all the papers that reference a relevant paper. Third, the authors of the relevant paper were checked, and their most recent publications on the same or similar topics were selected. Fourth, all these selected papers were analyzed using the exclusion procedure given in this section. In summary, three new papers (i.e., [25][28][43]) were identified as relevant to the research topic by the inclusion procedure.

A total of 15 papers were selected using the exclusion/inclusion procedure.

2.4 Class Model Slicing Techniques

The papers that were selected using the search strategy described in Section 2.3 were further analyzed, and eight primary studies were identified from these selected papers. Note that all the papers describing the same or similar technique were counted as one primary study. For example, papers, [40][41][42][43], describe the same slicing technique, and thus were counted as one primary study. In the remainder of this section we describe these eight primary studies on UML class model slicing:

1. Kagdi et al. [22] proposed a technique to slicing UML class models for software maintenance (e.g., to support the evolution of large software systems). The slic-

ing technique takes as input a large UML class model that represents a software system, and generates a slice, a subset of a UML class model, based on a slicing criterion. In their technique a slicing criterion includes could include classes, packages, components, operations and relationships (e.g., association, generalization, dependency) defined in the input class model. The computation of a slice starts from the model elements satisfying the slicing criterion and searches their adjacent model elements through the relationship defined in the UML metamodel. The search is terminated when the maximal path length between starting model elements and ending model elements is reached. The maximal path length is given by users as part of a slicing criterion. The slice includes all the model elements identified in the search. Their slicing technique can only handle basic UML class models, that is, class models without invariants and operation contracts.

2. Bae et al. [4][5] presented a slicing tool, *UMLSlicer*, for managing the complexity of the UML metamodel. Their tool can be used to decompose the UML metamodel into a set of metamodel fragments, where each metamodel fragment represents the structure of a UML diagram (e.g., sequence diagram). In their approach a slicing criterion includes a set of key classes given by a user. Their tool takes as input a slicing criterion, and produces a metamodel slice that includes a "Basic Slice" and an "Extended Slice". The "Basic Slice" consists of (1) the given classes, and (2) all the classes that are directly connected with the given classes through association relationship. The "Extended Slice" includes all the classes that are the direct and indirect ancestors of each metamodel element from the "Basic Slice". Their technique is domain dependent and therefore can only be used for slicing the UML metamodel.

3. Sen et al. [39] proposed a slicing technique for metamodel pruning (e.g., 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 and their dependent elements from the input metamodel. The dependency relationship between two elements is determined by a set of rules given in their paper. For example, one of their rules specifies that class A depends on class B if A is a subclass of B. The pruned metamodel also satisfies the structural constraints imposed by the input metamodel. Thus any instance of the pruned metamodel is an instance of the input metamodel. Their technique is domain independent and therefore can be used for slicing any metamodels that conform to the UML standard.

4. Blouin et al. [6][7] presented a domain specific language, *Kompren*, for developing model slicers. The tool that builds upon the *Kompren* language allows a user to provide a domain specific metamodel as input, and creates a slicer for the metamodel. The generated slicer takes as input a model that conforms to the metamodel, and produces a model fragment that is part of the input model. *Kompren* can be used to generate a variety of model slicers depending on the slicing purpose. For example, an *endogenous model slicer* ensures that the generated model fragments are valid instances of the metamodel used to build the slicer, while an *exogenous model slicer* relaxes the structural constraints (e.g., multiplicity constraint) specified on the input metamodel, and produces model fragments that conform to the modified metamodel. *Kompren* can generate tools for slicing basic UML class models and UML class models including OCL operation contracts or invariants.

5. Mall et al. [25][26] proposed a model slicing technique for a variety of purposes (e.g., model comprehension, impact analysis of design changes, critical model el-

ements identification). The inputs of their slicing technique include a class model and a sequence diagram, and produces a model dependency graph as an intermediate result. The model dependency graph is used to identify model elements that have dependency relationships with the elements involved in a given slicing criterion. A slicing criterion in their technique includes an object, one or multiple messages, and the initial model data that represents initialized values of attributes in the object during the execution of a scenario. The computation of a model slice requires the information from multiple models (i.e., class model and sequence diagram). This is in contrast to other works where slicing is performed on individual UML models. Note that their slicing technique cannot handle UML class models including OCL constraints.

6. Jeanneret et al. [20] used a slicing technique to estimate the footprint of an operation (i.e., the part of a UML class model used by an operation) without executing the operation. The generated footprint can be used for change impact analysis (i.e., an analysis involves checking if a modification on the elements contained by an operation's footprint have an impact on the operation). The slicing technique takes as input the contracts of a given operation (i.e., a slicing criterion) and a class model in which the operation is defined, and produces a metamodel footprint (i.e., a set of metamodel elements involved in the operation contract) as an intermediate result. The metamodel footprint is then used to guide the class model slicing process. The model elements that are not the instances of the elements involved in the metamodel footprint are removed from the input class model. Since an operation's contracts can be specified using the OCL, their technique can be used to slice UML class models including OCL operation contracts.

7. Lano et al. [27][28] described a slicing technique used to produce smaller UML class models for effective comprehension and analysis. Their slicing technique

reduces the size of a class model by removing attributes and OCL invariants from the controller class (i.e., a class that usually serves as the access point to the services of a system for external users) of the class model. A slicing criterion in their technique includes the OCL operation contracts defined on the controller class. The class slicing process follows two rules: (1) An OCL invariant that is not referenced by any operation contracts can be removed from the controller class; (2) an attribute that is not referenced by any operations or invariants can be removed from the controller class. A state machine is used in their slicing technique to determine if there exists a dependency between an operation and an attribute. Compared with other techniques, their technique focuses on slicing a single class rather than a class model.

8. Shaiky et al. [40][41][42][43] used a slicing technique to improve the scalability of an analysis that involves checking if a UML class model has a valid instance that satisfies the invariants defined in the class model. The slicing technique reduces the analysis problem of checking a large class model with OCL invariants into smaller subproblems, where each subproblem involves checking a model fragment with a subset of OCL invariants. Each model fragment is part of the input class model and can be analyzed separately. The OCL invariants are used as slicing criteria in their technique, and model elements that are not referenced by any invariants are removed from the input class model. The model decomposition process is guided by the following rule: All constraints restricting the same model element should be checked together and therefore must be contained in the same model fragment. Their slicing technique can handle only UML class models including OCL invariants.

Table 2.4: An Evaluation of Class Model Slicing Techniques

No.	Paper	PS	CMT	SC	IM	AT
1	[22]	Model maintenance	Basic	Class model elements	No	Automatable
2	[4][5]	Metamodel management	Basic	Classes	No	Automated
3	[39]	Metamodel pruning	Basic	Classes and properties	No	Automated
4	[6][7]	Building model slicer	Basic, Inv and Op	Class model elements	No	Automated
5	[25][26]	Model comprehension etc.	Basic	Classes, scenarios, and model data	Dependency graph	Automated
6	[20]	Change impact analysis	Op	Operation contracts	Metamodel footprint	Automated
7	[27][28]	Comprehension and analysis	InvOp	Operation contracts	No	Automatable
8	[40][41] [42][43]	Analysis	Inv	Invariants	Dependency graph	Automated

2.5 Evaluation

In this section we provide an evaluation of the class model slicing techniques described in Section 2.4. The evaluation aims to address the research questions proposed in Section 2.2. Table 2.4 shows the results of the evaluation:

1. Purpose: Five out of eight slicing techniques (1, 5, 6, 7, and 8) are used for model management (i.e., maintenance, comprehension, and analysis), two of them (2 and 3) are used for metamodel management (e.g., metamodel pruning), and only one technique (4) is used for building model slicers.

2. Class Model Type: Seven out of eight slicing techniques can handle only one type of class model, either basic class models (1, 2, 3 and 5) or non-basic class models (6 for class models including operation contracts, 8 for class models including invariants, and 7 for class models including both operation contracts and invariants),

while technique 4 can handle three types of class models. Note that metamodels are typically represented using UML class models, and thus techniques that handle metamodels can work for class models.

3. Slicing Criterion: Four out of eight techniques (1, 2, 3 and 4) use only class model elements in their slicing criteria, three of them operation contracts (6 and 7) and invariants (8) as their slicing criteria, and only one technique (5) uses elements from multiple models in its slicing criterion.

4. Intermediate Model: Only three out of eight techniques (5, 6, and 8) generates intermediate models in the model slicing process. Techniques 5 and 8 use dependency graphs to perform the dependency analysis, and technique 6 uses a metamodel footprint to extract an operation footprint from a class model.

5. Automation: Six out of eight techniques (2, 3, 4, 5, 6, and 8) are automated because they are supported by research prototypes, while techniques 1 and 7 are automatable since only slicing algorithms can be found in their papers.

2.6 Open Issues

One open issue regarding the existing class model slicing techniques is that many of them do not take invariants and operation contracts expressed in auxiliary constraint languages (e.g., OCL) into consideration when producing model slices. Their applicability is thus limited to situations in which the determination of slices does not require information found in constraints. This limits the utility of class model slicing techniques in (1) situations where invariants are needed to specify additional properties that cannot be expressed directly in a class model (e.g., acyclicity property), and (2) model-based software development approaches that are contract based (e.g., design by contract [30]). There are a few slicing techniques that take into consideration class model constraints

expressed in the OCL, but either they handle only invariants (e.g., technique 8) or only operation contracts (e.g., technique 6), or they can only be used to slice a single class (e.g., technique 7). Based on our systematic review, none of the above techniques can be used to slice class models including both invariants and operation contracts.

Another open issue is that many of the existing class model slicing techniques do not take multiple models into consideration. This limits the utility of class model slicing techniques in analyses where other types of models are needed. For example, consider an analysis that involves checking if an instance of a class model (i.e., an object model) satisfies the constraints defined in the class model. Such analysis takes as input both an object model and a class model. However, existing model slicing techniques do not take both an object model and a class model into account when producing model slices, and therefore cannot be used to improve the scalability of the above analysis.

Chapter 3

Background

In this dissertation we focus on two types of class model analysis: one that involves checking conformance between an object configuration and a class model with specified invariants, and the other that analyzes sequences of operation invocations to uncover invariant violations. Section 3.1 describes some model conformance checking tools that can be used to handle the first type of analysis. Section 3.2 describes an approach we developed to tackle the second type of class model analysis.

3.1 Class Model Conformance Checking Tools

In MDD, analysis that involves checking conformance between a class model with invariants and object configurations can be used to identify the design errors in the class model. For example, if an object configuration describes an invalid structure and is found to be consistent with a class model then there may exist loose constraints in the class model. The invariants defined in the class model may thus need to be strengthened. If an object configuration describes a valid structure and is inconsistent with a class model then there may exist overly restrictive invariants in the class model that need to be weakened. In the remainder of this section we describe some conformance checking tools.

USE [15][37], developed by the Database Systems Group at Bremen University, is

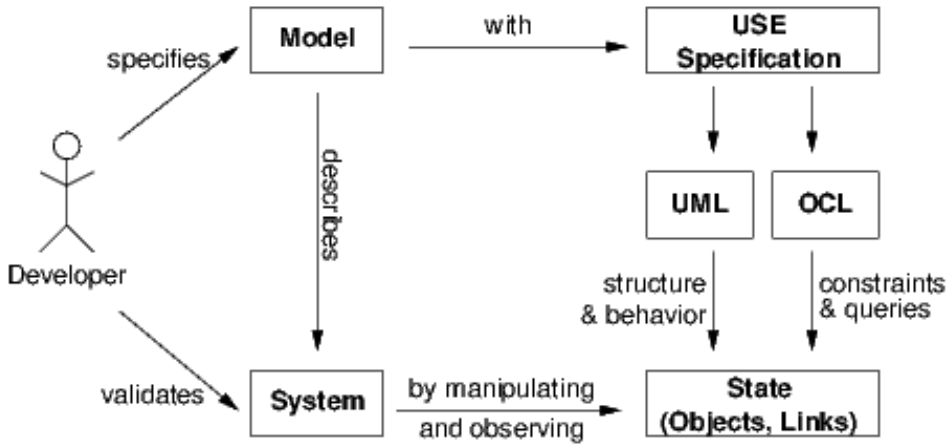


Figure 3.1: USE Tool Overview (excerpted from the USE project [15])

a modeling tool for specifying object-oriented systems. Figure 3.1 shows an overview of the USE tool. It allows developers to generate object configurations that are checked against user-specified properties (e.g, invariants) expressed in a class model. A system with a set of invariants can be specified using the USE specification language, that is based on a subset of the UML class model notation [13] and the Object Constraint Language (OCL) [44]. An object configuration 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 object configuration.

Alloy [19][18] is a formal specification language that was developed by the Software Design Group at MIT. It has very 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. The search space is typically bound by users in the form of limits on the number of entities to be included in the search space. 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 can be used to check conformance between an object configuration and a class model with invariants. For example, class models can be specified using Alloy signatures and fields, invariants defined on class models can be specified using Alloy facts, and object configurations can be specified using Alloy predicates. If the Alloy Analyzer cannot return an instance of an Alloy model for a predicate specifying an object configuration within a bounded scope, an object configuration specified using the predicate may not be consistent with a class model expressed using the Alloy model.

The Kermeta language [12][21][31] was developed by Triskell Team at INRIA. It is an executable metamodeling language implemented on top of the Eclipse Modeling Framework (EMF) [45] 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 [32]. The Kermeta workbench allows developers to specify well-formedness rules (i.e., invariants) on class models. These rules can be expressed using an OCL-like specification language. The Kermeta workbench provides several APIs for evaluating OCL-like invariants against object configurations. It reports warning information if a given object configuration does not satisfy the invariants defined in the class model.

The Eclipse OCL project [48] is an implementation of the OCL OMG standard [44] for EMF-based models. It provides APIs for (1) analyzing and transforming the abstract syntax model of OCL expressions, and (2) parsing and evaluating OCL constraints and queries on UML class models expressed in the EMF Ecore form [45]. The extensibility of the provided APIs allows software modelers to develop their own customized prototypes for a variety of OCL analysis tasks.

3.2 Analyzing Invariants and Operation Contracts of UML Class Models

In previous work [46] we developed a class model analysis approach that uses the Alloy Analyzer [18] to find scenarios (sequences of operation invocations) that start in valid states (states that satisfy the invariants in the class model) and end in invalid states (states that satisfy the negation of the invariants). The analysis uses the operation contracts to determine the effects operations have on the state. If analysis uncovers a sequence of operation calls that moves the system from a valid state to an invalid state, then the designer uses the trace information provided by the analysis to determine how the operation contracts should be changed to avoid this scenario.

The approach uses UML-to-Alloy and Alloy-to-UML transformations to shield the designer from the back-end use of the Alloy language and analyzer. The transformations used in the approach build upon the UML2Alloy transformation tool [8][2][1] developed at the University of Birmingham. The work proposed in [46] extends this prior work by providing support for transforming functional behavior specified in a UML class model to an Alloy model that specifies behavioral traces.

The approach in [46] also builds upon our previous work on the Scenario-based UML Design Analysis (ScUDA) approach [55][53][54][52]. A designer uses ScUDA to check whether a specific functional scenario is supported by a design class model in which operations are specified using the OCL. In ScUDA, the property to be verified is expressed as a specific sequence of state transitions (a functional scenario). The approach described in [46] goes further in that the property to be verified is expressed in terms of valid and invalid states, and analysis attempts to uncover scenarios that start in a specified valid state and end in a specified invalid state. In summary, ScUDA is used to answer the question “Is the given scenario supported by the UML class model?”, while the approach described in [46] is used to answer the question “Is there a scenario

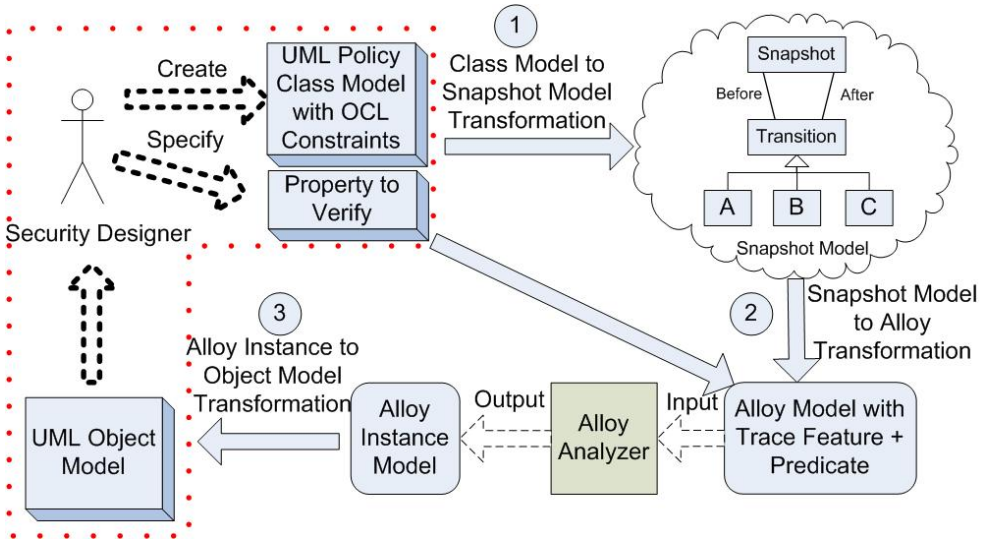


Figure 3.2: Class Model Analysis Approach Overview

supported by the UML class model that starts in a specified valid state and ends in a specified invalid state?".

Figure 3.2 shows an overview of the class model analysis approach. The dotted area includes the front-end activities and models. The front-end models are the only models that a security designer needs to manipulate directly. The security designer is responsible for 1) modeling access control policies using UML class model notation and the OCL, and 2) specifying the property-to-verify. The property-to-verify is expressed in terms of an invariant and its negation: The invariant characterizes the form of valid source states, and its negation characterizes target invalid states. Object configurations representing software states are called *snapshots* in this paper.

The back-end activities use three transformations (indicated in Fig. 3.2). Transformation 1 transforms the UML policy model to a class model, called a *snapshot transition model*, that specifies valid snapshot transitions, where a transition describes the effect of an operation on a state. The UML-to-snapshot model transformation defined in the SUDA approach [55] is used for this purpose. Transformation 2 converts the snapshot model to an Alloy model. The property-to-verify is transformed to an Alloy predicate,

referred to as the *verification predicate*, that is added to the Alloy model generated from the snapshot model. The resulting Alloy model is fed into the Alloy Analyzer and the verification predicate is evaluated. The Alloy Analyzer is used to determine if there exists an operation invocation sequence that starts from a specified valid snapshot and ends in a specified invalid snapshot. If the Analyzer finds a sequence then Transformation 3 is needed to convert the Alloy instance model of the sequence to a UML object model describing the sequence.

Chapter 4

The Slicing Techniques

In this chapter we propose two class model slicing techniques that can be used to decompose large class models into model fragments where each model fragment contains just those elements needed to perform the analysis. The first slicing technique can be used to improve the scalability of a class model analysis technique that involves checking conformance between an object configuration and a class model with specified invariants (Section 4.1) and the second slicing technique can be used to improve the scalability of a class model analysis technique that involves analyzing sequences of operation invocations to uncover invariant violations (Section 4.2).

In the reminder of this chapter we describe the first slicing technique in Section 4.1 and the second slicing technique in Section 4.2 respectively.

4.1 Co-slicing Class Model and Object Configuration

To improve the scalability of class model conformance analysis, we propose a tool-independent approach that involves reducing the size of the analysis inputs to make the analysis more efficient. The slicing approach is parameterized by slicing criteria, and can vary along two dimensions: Static versus dynamic slicing, and slicing based on single versus multiple invariants. Static slicing occurs when elements in the class model, but not elements in the object configuration, are used to determine how the class

model and object configuration are sliced. Dynamic slicing occurs when elements in the object configuration are used to slice the model and object configuration. This gives developers flexibility in determining the forms of domain knowledge used to break-up models and object configurations into fragments that can be more efficiently analyzed.

The proposed approach can have four slicing variations: static single invariant slicing, static multiple invariants slicing, dynamic single invariant slicing, dynamic multiple invariants slicing. We focus on two extreme variations in this dissertation:

- *Static Single Invariant Slicing:* In this case, a modeler is interested in checking the object configuration against a single invariant from a list of given invariants. The slicing technique analyzes the dependency between the invariant and the class model, and produces a class model fragment consisting only of elements that are referenced by the invariant. The class model fragment is then used to extract an object configuration fragment from the input object configuration. The object configuration fragment is an instance of the model fragment.

- *Dynamic Multiple Invariants Slicing:* This slicing technique uses the object configuration to identify a subset of given invariants that are applicable to the object configurations (referred to as *analysis relevant invariants*). The intuition behind this is based on the following observation: If each element in an object configuration is not an instance of any model element that is referenced by an invariant, the invariant can be removed from a list of invariants used to check the object configuration because the object configuration will not violate the invariant. The selected invariants are used to slice the input class model and the object configuration.

In the remainder of this section we present a UML class model and use the model to illustrate the above slicing variations.

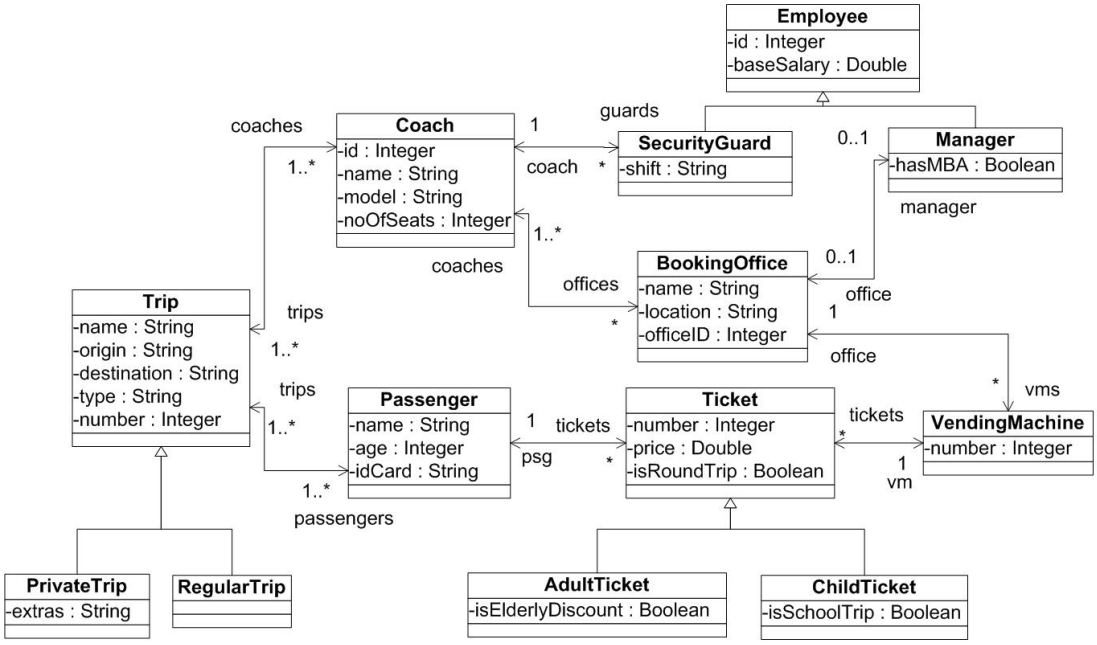


Figure 4.1: A UML class model that describes the information system of a bus company

4.1.1 A Class Model Example

Figure 4.1 shows part of a UML class model that describes the information system of a bus company (excerpted from [40]). In the *Coach* class model, a trip uses more than one coach, and a coach is controlled by multiple security guards. A passenger can buy multiple tickets from a vending machine located at 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 manger.

Invariants in the class model are specified using the OCL. Examples of OCL invariants for the *Coach* class model are given in Table 4.1.

Figure 4.2 shows an instance of the class model given in Figure 4.1. The object configuration in Figure 4.2 may or may not satisfy the invariants defined in the *Coach* class model. Tools such as USE can help modelers to determine if the object configuration is consistent with the invariants defined in the *Coach* class model.

Table 4.1: A list of invariants in the *Coach* model

// 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()→isUnique(t | t.number)
// Each passenger must have a non negative age. **Context** *Passenger* **inv NonNegativeAge:** self.age $\geq$ 0
// Each employee must have a unique number. **Context** *Employee* **inv UniqueEmployeeID:** Employee::allInstances()→isUnique(e | e.id)
// Each employee's base salary must be greater than 3000. **Context** *Employee* **inv BaseSalaryConstraint:** self.baseSalary $\geq$ 3000

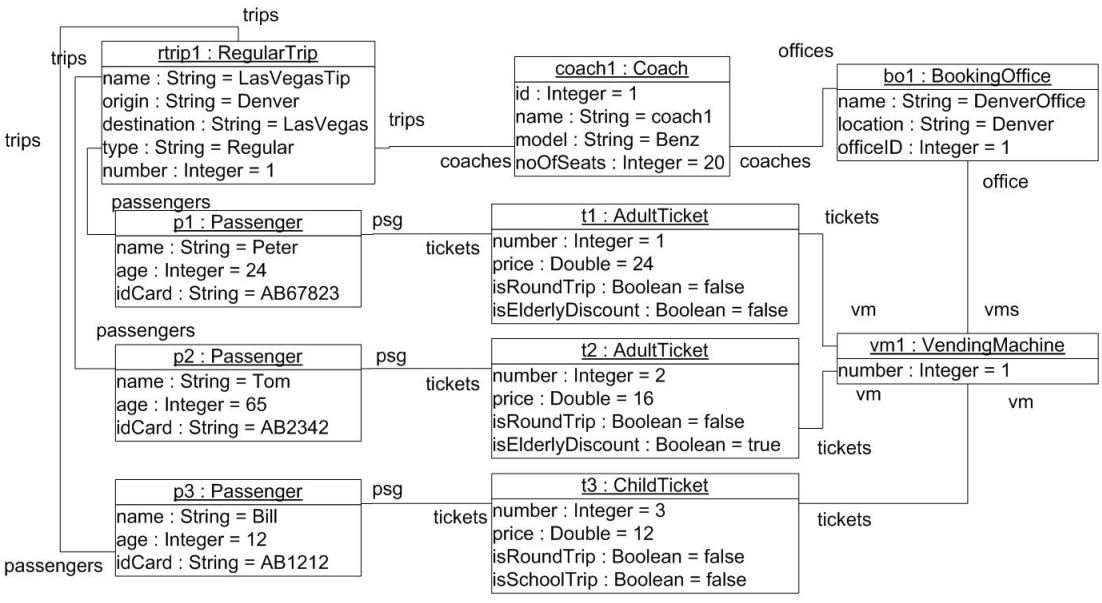


Figure 4.2: An object configuration conforming to the class model given in Figure 4.1

4.1.2 Static Single Invariant Slicing

Consider the case in which a modeler wants to check if the object configuration in Figure 4.2 satisfies the *MaxCoachSize* invariant defined in the class model in Figure 4.1. The static, single invariant slicing variation can be used for this purpose. The approach consists of two major steps. The first step uses Algorithm 1 to generate a class model fragment (i.e., the slice, see Figure 4.3a), that is, the part of the *Coach* class model that is referenced by the invariant. In the second step, Algorithm 2 is used to extract an object configuration fragment (i.e., the slice) from the object configuration in Figure 4.2, where each object element in the object configuration fragment is an instance of a class model element in the class model fragment generated by Algorithm 1.

Algorithm 1 first computes *DRMElmts*, a set of model elements that are directly referenced by the given invariant, and then computes *IRClss*, a set of classes that are indirectly referenced by the invariant. The generated class model fragment contains only elements that are from *DRMElmts* and *IRClss*.

In Algorithm 1, line 4 analyzes the dependencies between an invariant and a class model, and the dependency analysis is performed by traversing the syntax tree of the OCL invariant. For example, consider the analysis of the *MaxCoachSize* invariant. The *MaxCoachSize* invariant is defined in the context of class *Coach*, and thus directly 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 direct 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 direct 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 directly depends on attribute *noOfSeats* and its containing class, *Coach*. The

Algorithm 1: Extract a class model fragment

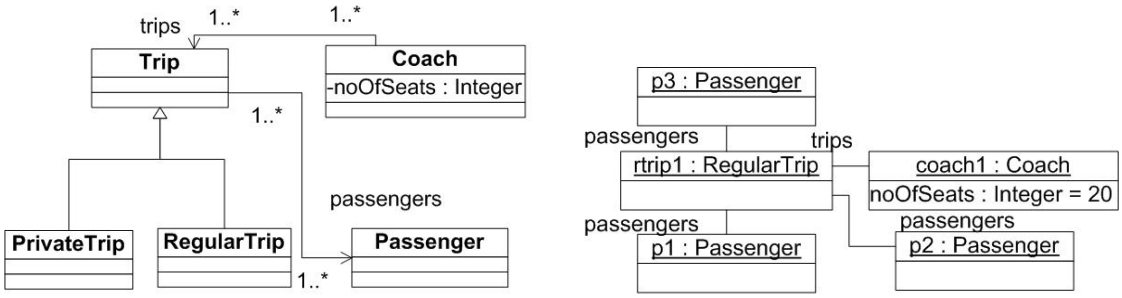
1: Input: A class model, *M*, and an invariant, *INV*
2: Output: A class model fragment
3: Algorithm Steps:
4: Analyze the dependencies between *M* and *INV*, set *DRMElmts* = a set of model elements (including classes, enumerations, attributes, and references) that are directly referenced by *INV*;
5: Set *IRClss* = {};
6: **for** each element, *ME*, in *DRMElmts* **do**
7: **if** *ME* is a class **then**
8: **if** *ME* has subclasses **then**
9: **for** each indirect and direct subclass, *subME*, of *ME* **do**
10: *IRClss* = *IRClss* ∪ *subME*;
11: **end for**
12: **end if**
13: **end if**
14: **end for**
15: Set *RMElmts* = *DRMElmts* ∪ *IRClss*;
16: Return *RMElmts*;

analysis thus reveals the invariant references and thus directly depends on the following class model elements: *Coach*, *Trip*, *Passenger*, *trips*, *passengers* and *noOfSeats*.

Lines 5-14 compute a set of classes that are indirectly referenced by the given invariant, where each class is a subclass of a class that is directly referenced by the invariant. For example, class *Trip* has two subclasses, *PrivateTrip* and *RegularTrip*. Since the *MaxCoachSize* invariant directly depends on *Trip*, there is thus an indirect dependency with *PrivateTrip* and *RegularTrip*.

In summary the class model elements that are referenced by the *MaxCoachSize* invariant include *Coach*, *Trip*, *PrivateTrip*, *RegularTrip*, *Passenger*, *trips*, *passengers* and *noOfSeats*. Thus a class model fragment that contains only these model elements is produced by Algorithm 1 (see Figure 4.3a).

Algorithm 2 uses the generated model fragment to extract an object configuration from the object configuration given in Figure 4.2 (see Figure 4.3b). The algorithm



(a) Class Model Fragment (b) Object Configuration Fragment

Figure 4.3: Sliced Class Model and Object Configuration

checks each object (see lines 5-7) in the given object configuration, and removes an object if the object's metaclass is not in the set of classes involved in the given class model fragment. For example, object *bo1:BookingOffice* in Figure 4.2 is not an instance of any class involved in the class model fragment in Figure 4.3a, and thus it can be removed from the object configuration.

Algorithm 2 also checks the slots and link ends of each unremoved object. Lines 9-13 removes a slot of an object if the slot's corresponding attribute is not included in *Attrs*, a set of attributes in the class model fragment. Lines 14-18 removes 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 4.2 has 5 slots and 4 link ends. Since class *RegularTrip* and its super class *Trip* in Figure 4.3a have no attributes, the slicing algorithm removed all the slots from object *rtrip1:RegularTrip* as indicated in Figure 4.3b. Similarly, class *RegularTrip* and its super class *Trip* in Figure 4.3a have only one reference, *passengers*. Thus the slicing algorithm removed link end *coaches* from object *rtrip1:RegularTrip*.

4.1.3 Dynamic Multiple Invariants Slicing

In dynamic, multiple invariants slicing dependencies among the class model, object configuration and invariants are used to determine the analysis relevant invariants,

Algorithm 2: Extract an object configuration fragment

```
1: Input: An object configuration, OC, and a class model fragment, CMF
2: Output: An object configuration fragment that conforms to CMF
3: Algorithm Steps:
4: Set Clss = a set of classes in CMF, set Attrs = a set of attributes in CMF, set Refs =
   a set of references in CMF;
5: for each object, obj, in OC do
6:    if obj's metaclass not in Clss then
7:       Remove obj from OC;
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 OC;
```

that is invariants with elements referenced by the object configuration. These relevant invariants are then used to slice the class model (*CM*) and object configuration (*OC*).

Algorithm 3 is used to identify invariants that refer to elements in the object configuration. Lines 5-7 compute *OCRClss*, a set of classes that are referenced by the objects in the input object configuration (*OC*). For example, *OCRClss* for the object configuration given in Figure 4.2 includes *RegularTrip*, *Passenger*, *Coach*, *AdultTicket*, *ChildTicket*, *BookingOffice* and *VendingMachine*.

Lines 8-13 check if each member of *INVList* is relevant w.r.t. the slicing of the class model and object configuration. Specifically, line 9 computes *INVRClss*, a set of classes that are directly and indirectly referenced by a given invariant, *INV*. The indirectly referenced classes are determined using lines 8-12 of Algorithm 1. For example, Table 4.2

Algorithm 3: Identify analysis relevant invariants

1: Input: An object configuration, *OC*, a class model, *CM*, and a list of invariants, *INVList*
2: Output: A list of invariants that are relevant for the analysis
3: Algorithm Steps:
4: Set *OCRClss* = {};
5: **for** each object, *obj*, in *OC* **do**
6: *OCRClss* = *OCRClss* ∪ *obj*'s metaclass
7: **end for**
8: **for** each invariant, *INV*, in *INVList* **do**
9: Set *INVRClss* = a set of classes that are directly and indirectly referenced by *INV*;
10: **if** *INVRClss* ∩ *OCRClss* returns ∅ **then**
11: Remove *INV* from *INVList*
12: **end if**
13: **end for**
14: Return *INVList*;

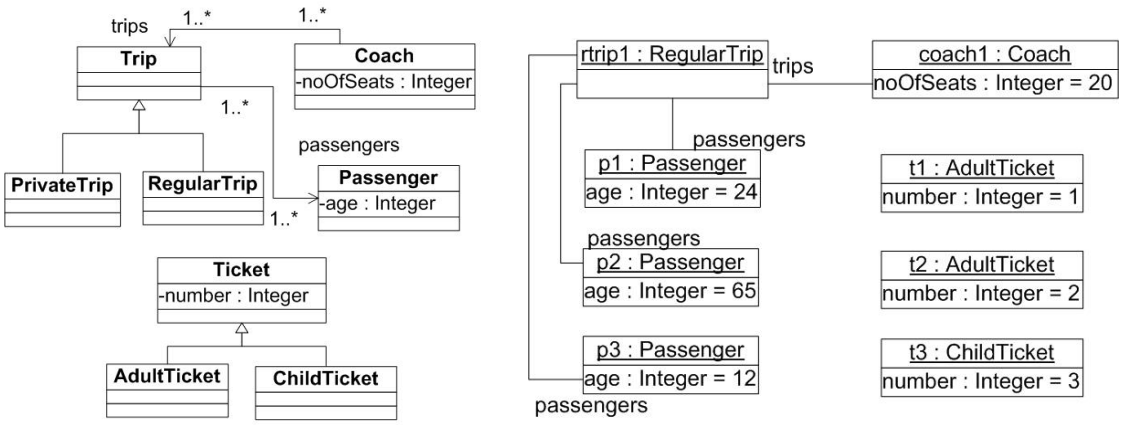
Table 4.2: Referenced classes for each invariant in the *Coach* model

Invariant	Directly Referenced Classes	Indirectly Referenced Classes
MinCoachSize	Coach	None
MaxCoachSize	Coach, Trip, Passenger	PrivateTrip, RegularTrip
UniqueTicketNumber	Ticket	AdultTicket, ChildTicket
NonNegativeAge	Passenger	None
UniqueEmployeeID	Employee	SecurityGuard, Manager
BaseSalaryConstraint	Employee	SecurityGuard, Manager

lists the referenced classes for each invariant defined in the *Coach* model.

Lines 10-12 identify *INV* as irrelevant if the intersection of *INVRClss* and *OCRClss* returns null. For example, both *UniqueEmployeeID* and *BaseSalaryConstraint* reference *Employee*, *Security Card*, and *Manager*, and thus *INVRClss* of *UniqueEmplyeeID* or *BaseSalaryConstraint* has no common element with *OCRClss* of the object configuration in Figure 4.2. Therefore, both *UniqueEmployeeID* and *BaseSalaryConstraint* are identified as irrelevant.

Algorithm 3 returns a list of invariants that are relevant for the analysis. These



(a) Class Model Fragment (b) Object Configuration Fragment

Figure 4.4: Sliced Class Model and Object Configuration

invariants are then used to slice *CM* and *OC*. Specifically the slicing technique first applies a modified version of Algorithm 1 to *CM*, and generates a class model fragment that contains only elements that are referenced by the analysis relevant invariants. The change to Algorithm 1 involves (1) replacing a single invariant *INV* with a list of invariants *INVList*, and (2) setting *DRMElmts* as a set of model elements (including classes, enumerations, attributes, references) that are directly referenced by *INVList*. The slicing technique then applies Algorithm 2 to *OC*, and produces an object configuration fragment that is an instance of the generated class model fragment. For example, Figure 4.4a shows a class model fragment that is referenced by a list of analysis relevant invariants (*MinCoachSize*, *MaxCoachSize*, *UniqueTicketNumber*, and *NonNegativeAge*), and Figure 4.4b shows an object configuration that is an instance of the class model fragment in Figure 4.4a.

4.2 Slicing a Class Model with OCL Constraints

The model slicing technique described in this section is used to decompose a large class model into fragments, where each fragment contains model elements needed to analyze

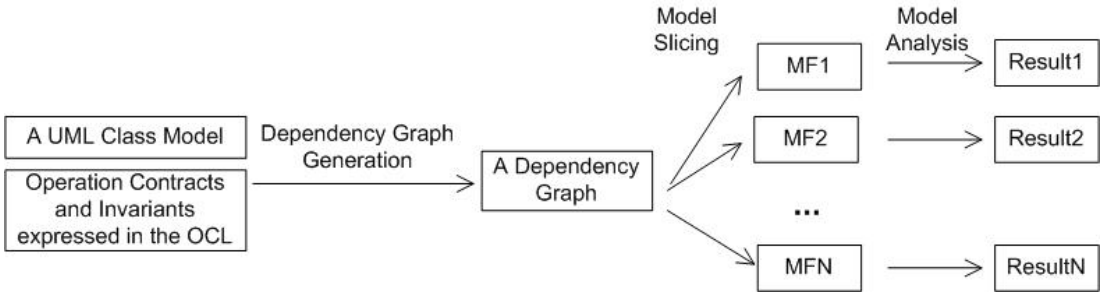


Figure 4.5: Technique Overview

a subset of the invariants and operation contracts in the class model. Fig. 4.5 shows an overview of the slicing technique.

The input to the technique is a UML class model with invariants and operation contracts expressed in the OCL. The technique has two major steps. In the first step, the input class model with OCL constraints is analyzed to produce a dependency graph that relates (1) invariants to their referenced model elements, and (2) operation contracts to their containing classes and other referenced classes and class properties. The dependencies among model elements are determined by relationships defined in the UML metamodel.

In the second step of the technique, the dependency graph is used to generate slicing criteria, and the criteria are then used to extract one or more model fragments from the class model. The generated model fragments can be analyzed separately.

In the remainder of this section we present a UML class model and use the model to illustrate the process for generating a dependency graph and the slicing algorithm used to decompose the class model into model fragments.

4.2.1 Illustrating Example

We will use the Location-aware Role-Based Access Control (LRBAC) model, proposed by Ray et al. [34] [35] [36], to illustrate the model slicing technique. LRBAC is an extension of Role-Based Access Control (RBAC) [38] that takes location into consider-

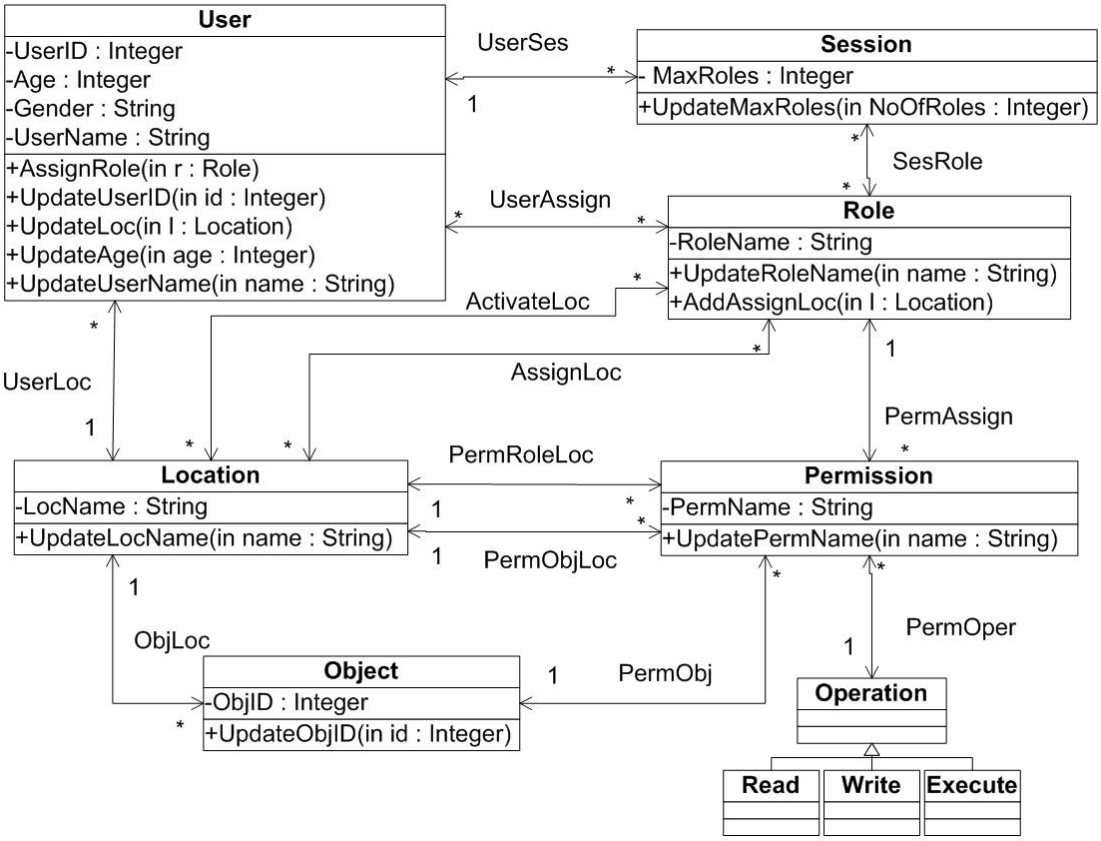


Figure 4.6: A Partial LRBAC Class Model

ation when determining whether a user has permission to access a protected resource.

In LRBAC, roles can be assigned to, or deassigned from users. A role can be associated with a set of locations in which it can be assigned to, or activated by users. A role that is associated with locations can be assigned to a user only if the user is in a location in which the role can be assigned. A user can create a session and activate his assigned roles in the session. A role can be activated in a session only if the user that creates the session is in a location in which the role can be activated. Figure 4.6 shows part of a design class model that describes LRBAC features.

Permissions are granted to roles, and determine the resources (*objects*) that a user can access (*read*, *write* or *execute*) via his activated roles. Permissions are associated with locations via two relationships: *PermRoleLoc* and *PermObjLoc*. *PermRoleLoc* links a

Table 4.3: A list of operation contracts in the *LRBAC* model

// Op1: Assign a role *r* to user *u* **Context** User::AssignRole(r:Role) // Precondition: user *u* has not been assigned role *r* and user *u* is in a location in // which role *r* can be assigned to him **Pre:** self.UserAssign→excludes(r) and r.AssignLoc→includes(self.UserLoc) // Postcondition: user *u* has been assigned role *r* **Post:** self.UserAssign = self.UserAssign@pre→including(r)
// Op2: Update a user's ID **Context** User::UpdateUserID(id:Integer) **Pre:** self.UserID != id **Post:** self.UserID = id
// Op3: Move a user into a new location *l* **Context** User::UpdateLoc(l:Location) // Precondition: user *u* has not been in location *l* and user *u* has not been assigned any role **Pre:** self.UserLoc→excludes(l) and self.UserAssign→isEmpty() // Postcondition: user *u* has been in location *l* **Post:** self.UserLoc→includes(l)
// Op4: Update a user's age **Context** User::UpdateAge(age:Integer) **Pre:** age > 0 **Post:** self.Age = age
// Op5: Update a user's name **Context** User::UpdateUserName(name:String) **Pre:** self.UserName != name **Post:** self.UserName = name

permission to its set of allowable locations for the role associated with the permission, and *PermObjLoc* links a permission to its set of allowable locations for the object associated with the permission. *MaxRoles* in class *Session* refers to the maximum number of roles that can be activated by a session.

Operation contracts and invariants in the LRBAC model are specified using the OCL. Examples of OCL operation contracts for the LRBAC model are given in Table 4.3 and Table 4.4. Examples of OCL invariants for the LRBAC model are given in Table 4.5.

For the LRBAC model, one may want to determine if there is a scenario in which the operation contracts allow the system to move into a state in which a user has unauthorized access to resources. In previous work [46], we developed a class model analysis

Table 4.4: A list of operation contracts in the *LRBAC* model

// Op6: Update a session's maximum activated roles **Context** Session::UpdateMaxRoles(NoOfRoles:Integer) **Pre:** self.MaxRoles != NoOfRoles **Post:** self.MaxRoles = NoOfRoles
// Op7: Update a role's name **Context** Role::UpdateRoleName(name:String) **Pre:** self.RoleName != name **Post:** self.RoleName = name
// Op8: Add an *AssignLoc* link between a role and a location **Context** Role::AddAssignLoc(l:Location) **Pre:** self.AssingLoc→excludes(l) **Post:** self.AssignLoc = self.AssignLoc@pre→including(l)
// Op9: Update a location's name **Context** Location::UpdateLocName(name:String) **Pre:** self.LocName != name **Post:** self.LocName = name
// Op10: Update a permission's name **Context** Permission::UpdatePermName(name:String) **Pre:** self.PermName != name **Post:** self.PermName = name
// Op11: Update an object's ID **Context** Object::UpdateObjID(id:Integer) **Pre:** self.ObjID != id **Post:** self.ObjID = id

technique that uses the Alloy Analyzer [18] to find scenarios (sequences of operation invocations) that start in valid states (states that satisfy the invariants in the class model) and end in invalid states. The analysis uses the operation contracts to determine the effects operations have on the state. If analysis uncovers a sequence of operation calls that moves the system from a valid state to an invalid state, then the designer uses the trace information provided by the analysis to determine how the operation contracts should be changed to avoid this scenario. Like other constraint solving approaches, performance degrades as the size of the model increases. The slicing technique described in the paper can improve the scalability of the analysis approach by reducing the problem to one of separately analyzing smaller model fragments.

Table 4.5: A list of invariants in the *LRBAC* model

// Inv1: Each user's age must be greater than 0. **Context** *User* **inv NonNegativeAge:** self.Age $\geq$ 0
// Inv2: Each user has a unique ID. **Context** *User* **inv UniqueUserID:** User.allInstances()→forAll(u1, u2:User|u1.UserID = u2.UserID implies u1 = u2)
// Inv3: Each user is either male or female. **Context** *User* **inv GenderConstraint:** self.Gender = "male" or self.Gender = "female"
// Inv4: For every role *r* that is assigned to a user, the user's location belongs to // the set of locations in which role *r* can be assigned. **Context** *User* **inv CorrectRoleAssignment:** self.UserAssign→forAll(r|r.AssignLoc→includes(self.UserLoc))
// Inv5: The number of roles a user can activate in a session cannot exceed the value // of the session's attribute, *MaxRoles*. **Context** *Session* **inv MaxActivatedRoles:** self.MaxRoles >= self.SesRole→size()
// Inv6: Each object has a unique ID. **Context** *Object* **inv UniqueObjectID:** Object.allInstances()→forAll(o1, o2:Object|o1.ObjID = o2.ObjID implies o1 = o2)

4.2.2 Constructing a Dependency Graph

Dependencies among invariants, operation contracts and model elements are computed by traversing the syntax tree of the OCL invariants and operation contracts. For example, consider the analysis of the operation contract for *AssignRole* (the contract is given in Section 2). The expression *self*.*UserAssign* is an association end call expression and it returns a set of roles assigned to the user (referred to by *self*). There is thus a dependency between this contract and the class *Role*. The expression *self*.*UserLoc* returns a user's current location, and thus there is a dependency with the class *Location*. The parameter *r* refers to an instance of class *Role*, and *r*.*AssignLoc* returns a set of locations in which role *r* can be assigned to any user. The analysis thus reveals the operation contract for *AssignRole* references and thus depends on, the following classes: *User*, *Role* and *Location*. If an OCL constraint involves a statement like *Role*.*allInstances*(), then the

Table 4.6: Referenced Classes and Attributes for Each Operation Contract/Invariant Defined in the LRBAC model

Operation Contract/Invariant	Referenced Classes	Referenced Attributes
Op1 AssignRole	User, Role, Location	None
Op2 UpdateUserID	User	UserID
Op3 UpdateLoc	User, Role, Location	None
Op4 UpdateAge	User	Age
Op5 UpdateUserName	User	UserName
Op6 UpdateMaxRoles	Session	MaxRoles
Op7 UpdateRoleName	Role	RoleName
Op8 AddAssignLoc	Role, Location	None
Op9 UpdateLocName	Location	LocName
Op10 UpdatePermName	Permission	PermName
Op11 UpdateObjID	Object	ObjID
Inv1 NonNegativeAge	User	Age
Inv2 UniqueUserID	User	UserID
Inv3 GenderConstraint	User	Gender
Inv4 CorrectRoleAssignment	User, Role, Location	None
Inv5 MaxActivatedRoles	Session, Role	MaxRoles
Inv6 UniqueObjectID	Object	ObjID

OCL constraint references class Role. A similar analysis is done for each OCL contract and invariant. Table 4.6 lists the referenced classes and attributes for the contracts and invariants defined in the LRBAC model.

The computed dependencies and relationships defined in the UML metamodel are used to build a dependency graph. A dependency graph consists of nodes and edges, where each node represents a model element (e.g., classes, attributes, operations and invariants), and each edge represents a dependency between two elements. For example, if a class model has only one class that includes only one attribute, the generated dependency graph consists of two nodes, a node representing the class and a node representing the attribute, and one edge that represents the relationship between the attribute and its containing class.

Figure 4.7 shows a graph that describes the dependency relationship among classes, attributes, operations and invariants of the LRBAC class model described in Fig. 4.6.

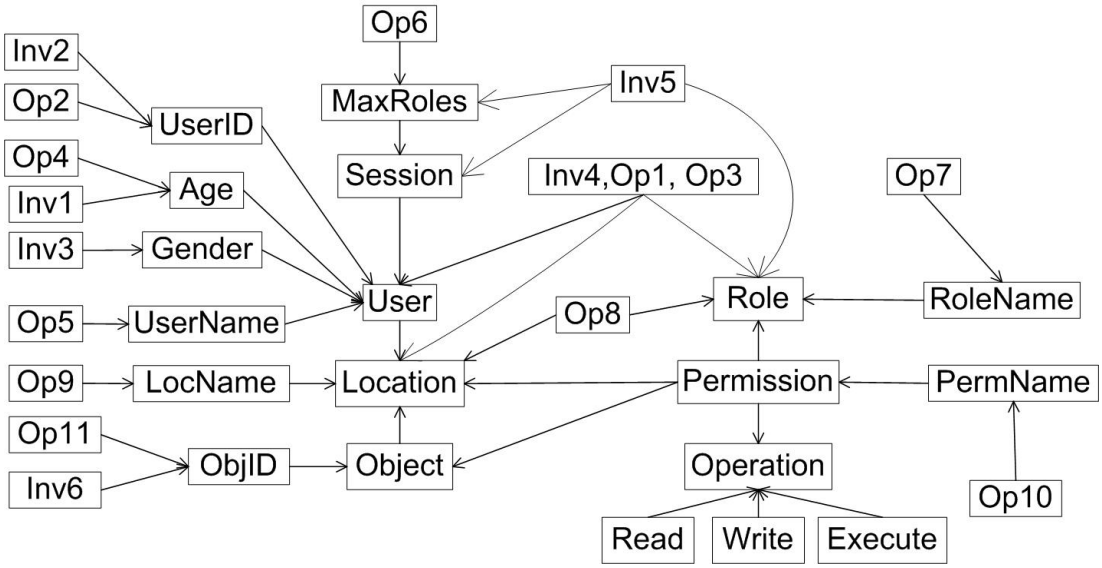


Figure 4.7: A Dependency Graph

Algorithm 4 describes the process used to generate the graph.

Steps 1 to 5 describe how the metamodel relationships and computed dependencies between OCL invariants and contracts and their referenced model elements are used to build an initial dependency graph. In step 6 of the algorithm, if an operation contract (*op*) or invariant (*inv*) only references its context class, *cls*, and an attribute in *cls*, *attr*, the edge that points to vertex *cls* from vertex *op* (or *inv*), can be removed because the dependency can be inferred from the dependency between the vertex *cls* and the vertex *attr*. For example, Table 4.6 shows that operation *UpdateUserID*'s ($Op2$) only references class *User* and its attribute *UserID* in its specification. The edge pointing to vertex *User* from vertex $Op2$ is redundant, and is thus removed from the dependency graph shown in Fig. 4.7. Invariant $Inv5$ in Table 4.6 references its context class, *Session*, and class *Session*'s attribute, *MaxRoles*, in the specification. But the edge pointing to vertex *Session* from vertex $Inv5$ cannot be removed from Fig. 4.7 since invariant $Inv5$ also references class *Role* in its specification through the navigation from class *Session* to class *Role*.

Algorithm 4: Dependency Graph Generation Algorithm

Input: A UML Class Model + OCL Operation Contracts/Invariants
Output: A Dependency Graph
Algorithm Steps:
Step 1. Create a vertex for each class, attribute, operation contract and invariant of the class model in the dependency graph.
Step 2. For every attribute, *attr* defined in a class, *cls*, create a directed edge from vertex *attr* to vertex *cls*.
Step 3. For every class, *sub*, that is a subclass of a class, *super*, create a directed edge from vertex *sub* to vertex *super*.
Step 4. For every class that is part of a container class (i.e., a class in a composition relationship), create a directed edge to a container class vertex from a contained class vertex.
Step 5. If there is an association between class *x* and *y*, and the lower bound of the multiplicity of the association end in *y* is equal to or greater than 1, create a directed edge to vertex *y* from vertex *x*.
Step 6. For every referenced class (or attribute, *attr*), *cls*, of an operation contract (or invariant, *inv*), *op*, create a directed edge to vertex *cls* (or *attr*) from vertex *op* (or *inv*). If the operation contract (or invariant) only references its context class, *cls*, and its context class's attribute (or attributes) in its specification, the edge that points to vertex *cls* from vertex *op* (or *inv*), is removed.

4.2.3 Analyzing a Dependency Graph

The generated dependency graph is used to guide the decomposition of a model into fragments that can be analyzed separately. The first step is to identify model elements that are not involved in the analysis. These are referred to as *irrelevant model elements*. The intuition behind this step is based on the following observation: If the classes and attributes that are referenced by an operation, are not referenced by any invariant, the operation as well as its referenced classes and attributes (i.e., analysis-irrelevant model elements) can be removed from the class model because a system state change triggered by the operation invocation will not violate any invariant defined in the model. Similarly, if the classes and attributes that are referenced by an invariant, are not referenced by any operation, the invariant as well as its referenced classes and attributes (i.e., analysis-irrelevant model elements) can be removed from the class model because any operation

invocation that starts in a valid state will not violate the invariant. *Irrelevant model elements* are identified using the process described in Algorithm 5, and are removed from the class model.

The second step is to identify model elements that are involved in a *local* analysis problem. A *local* analysis problem refers to an analysis that can be performed within the boundary of a class [40]. Model elements that are involved in a *local* analysis problem are referred to as *local analysis model elements*. For example, operation *UpdateUserID* in Fig. 4.6 is used to modify the value of attribute *UserID* in class *User*, and invariant *Inv2* defines the uniqueness constraint on *UserID*. The invocation of operation *UpdateUserID* may or may not violate the constraint specified in *Inv2*, but it will not violate other invariants because *UserID* is not referenced by other operations or invariants. Thus an analysis that involves checking if an invocation of *UpdateUserID* violates *Inv*2 can be performed within the boundary of *User*. Model elements that are involved in *local* analysis problems are identified using the process described in Algorithm 6. Note that in the above example, *UpdateUserID*, *UserID* and *Inv2* are identified *local analysis model elements*. Thus these model elements and their dependent model elements (*User* and *User*'s dependent class *Location*) can be extracted from the LRBAC model and analyzed separately.

In the third step, the class model is further decomposed into a list of model fragments using Algorithm 7.

4.2.3.1 Identifying *Irrelevant Model Elements*:

Algorithm 5 is used to remove analysis-irrelevant model elements. The algorithm first computes *ARClsAttrVSet*, a set of analysis relevant class and attribute vertices, where each vertex is directly dependent on at least one operation vertex and at least one invariant vertex. The algorithm then computes *AROpInvVSet*, a set of analysis relevant operation and invariant vertices, where each vertex has a directly dependent vertex that

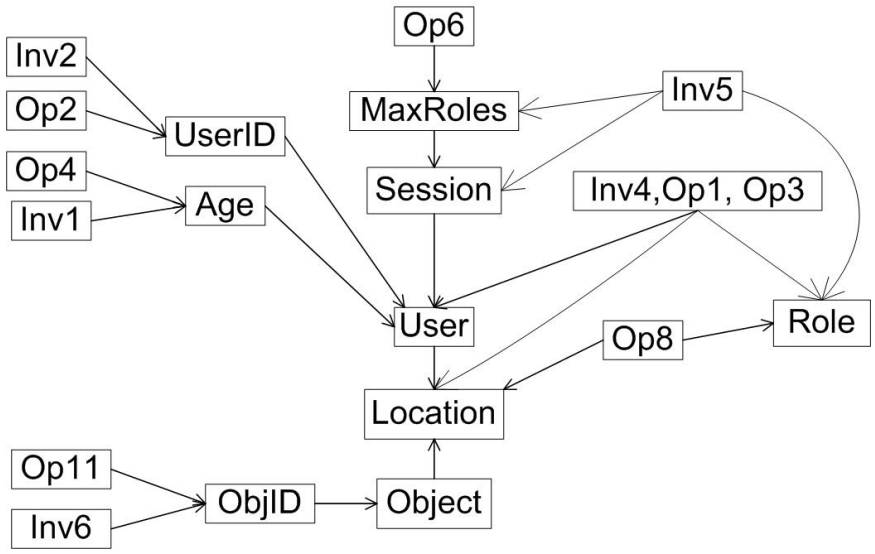


Figure 4.8: A Dependency Graph Representing a LRBAC Model with the *Irrelevant Model Elements* Removed

belongs to *ARClsAttrVSet*. The algorithm then performs a Depth-First Search (DFS) from each vertex in *AROpInvVSet*, and labels all the analysis-relevant vertices, *ARVSet*. The vertices not in *ARVSet* represent the *irrelevant model elements* that need to be removed from the class model.

Figure 4.8 shows a dependency graph representing a LRBAC model with the analysis irrelevant model elements removed. Lines 6-14 of Algorithm 5 compute *ARClsAttrVSet*. Lines 6-9 compute *OpDVSet*, a set of directly dependent attribute and class vertices from each operation vertex. Similarly, lines 10-13 compute *InvDVSet*, a set of directly dependent attribute and class vertices from each invariant vertex. *ARClsAttrVSet* is the intersection of *OpDVSet* and *InvDVSet*.

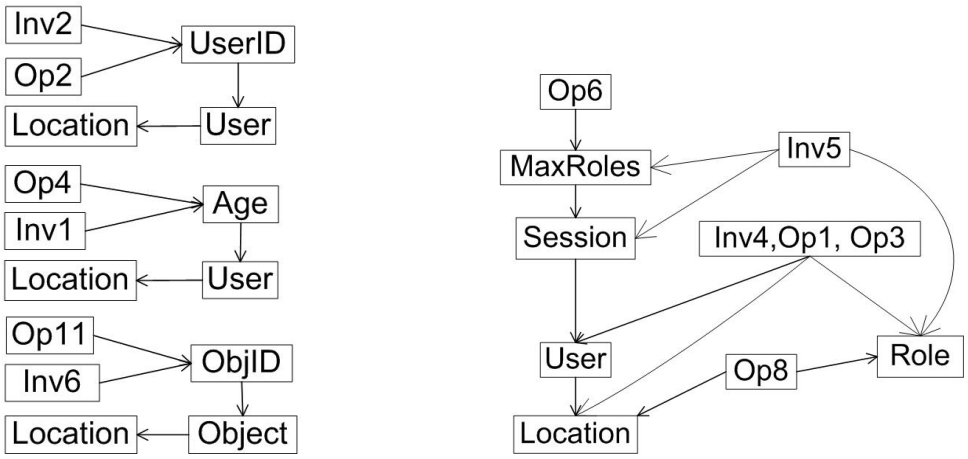
Lines 15-19 compute *AROpInvVSet*. For example, vertex $Op4$ is an analysis-relevant operation vertex because its directly dependent vertex, *Age*, is an analysis-relevant attribute vertex, while vertex $Op5$ is analysis-irrelevant because *UserName* is not an analysis-relevant vertex. Lines 21-25 compute *ARVSet*.

Algorithm 5: Irrelevant Model Elements Identification Algorithm

1: Input: A dependency graph
2: Output: A set of analysis-irrelevant vertices
3: Algorithm Steps:
4: Set *OpVSet* = a set of operation vertices, *InvVSet* = a set of invariant vertices;
5: Set *VSet* = all the vertices in the dependency graph, *OpDVSet* = {}, *InvDVSet* = {};
6: **for** each operation vertex *OpV* in *OpVSet* **do**
7: Get a set of *OpV*'s directly dependent vertices, *OpVDDSet*;
8: *OpDVSet* = *OpDVSet* ∪ *OpVDDSet*;
9: **end for**
10: **for** each invariant vertex *InvV* in *InvVSet* **do**
11: Get a set of *InvV*'s directly dependent vertices, *InvVDDSet*;
12: *InvDVSet* = *InvDVSet* ∪ *InvVDDSet*;
13: **end for**
14: Set *ARClsAttrVSet* = *OpDVSet* ∩ *InvDVSet*, Set *AROpInvVSet* = {};
15: **for** each vertex *V* in *OpVSet* ∪ *InvVSet* **do**
16: **if** one of *V*'s directly dependent vertex is in *ARClsAttrVSet* **then**
17: *AROpInvVSet* = *AROpInvVSet* ∪ *V*; Break;
18: **end if**
19: **end for**
20: Set *ARVSet* = {};
21: **for** each vertex *V* in *AROpInvVSet* **do**
22: Perform a Depth-First Search (DFS) from vertex *V*;
23: Get a set of labeling vertices, *VDFSSet*, from vertex *V*'s DFS tree;
24: *ARVSet* = *ARVSet* ∪ *VDFSSet*;
25: **end for**
26: Return (*VSet* - *ARVSet*);

4.2.3.2 Identifying *Local Analysis Model Elements*:

Algorithm 6 is used to identify fragments representing local analysis problems. The algorithm first computes a set of attribute and class vertices, *LocalVSet*, that are involved in the local analysis problems. A vertex, *ClsAttrV*, is added to *LocalVSet* only if (1) the vertex is a member of *ARClsAttrVSet* (indicated by Line 6), and (2) all vertices directly dependent on *ClsAttr* have no other directly dependent vertices (indicated by Lines 7-15). The algorithm then uses the vertices in *LocalVSet* to construct new dependency graphs, where each graph represents a model fragment involved in a local analysis prob-



(a) Local Analysis Model Elements identified by Algorithm 6

(b) Reduced Dependency Graph with Local Analysis Model Elements Removed

Figure 4.9: Dependency Graphs Representing Model Fragments Extracted from the LRBAC Model in Fig. 4.6

lem.

The dependency graph in Fig. 4.8 is decomposed into several subgraphs, as shown in Fig. 4.9a and Fig. 4.9b, using Algorithm 6. Each dependency graph in Fig. 4.9a represents a model fragment involved in a local analysis problem.

For example, $\{UserID, Age, ObjID\}$ is the *LocalVSet* set of the dependency graph in Fig. 4.8. The vertices that directly depend on vertex *ObjID* are $Op11$ and $Inv6$, and they are moved from *DG* to a new dependency graph. Vertex *ObjID*'s DFS tree consists of *ObjID*, *Object* and *Location*, and they are copied from *DG* to the new dependency graph. *ObjID* is then removed from *DG*. Note that vertex *Object* becomes analysis-irrelevant in *DG* after vertex *ObjID*, $Op11$, and $Inv6$ have been removed from *DG*. Thus it is necessary to perform Algorithm 5 on *DG* to remove the analysis-irrelevant vertices, as indicated by Line 25.

4.2.3.3 Decomposing the Dependency Graph:

Algorithm 7 is used to decompose a dependency graph without analysis-irrelevant vertices and local analysis problem related vertices. The algorithm computes a set of

Algorithm 6: Local Analysis Problem Identification Algorithm

```
1: Input: A dependency graph, DG, produced from the original graph after removing
   the irrelevant vertices produced by Algorithm 5
2: Output: A set of dependency graphs
3: Algorithm Steps:
4: Reuse ARClsAttrVSet in Algorithm 5;
5: Set LocalVSet = {};
6: for each vertex, ClsAttrV, in ARClsAttrVSet do
7:    Set Flag = TRUE;
8:    for each vertex, V, that is directly dependent on ClsAttrV do
9:       if V has other directly dependent vertices then
10:         Set Flag = FALSE; Break;
11:      end if
12:   end for
13:   if Flag == TRUE then
14:      LocVSet = LocVSet ∪ ClsAttrV;
15:   end if
16: end for
17: for each vertex, LocalV, in LocalVSet do
18:   Create an empty dependency graph, SubDG;
19:   Move the operation and invariant vertices that directly depend on LocalV, from
      DG to SubDG;
20:   Perform a DFS from vertex LocalV;
21:   Get a set of labeling vertices, LocalVDFSSet, from vertex LocalV's DFS tree;
22:   Copy LocalVDFSSet from DG to SubDG;
23:   Remove LocalV from DG;
24: end for
25: Perform Algorithm 5 on DG to remove analysis-irrelevant vertices;
```

slicing criteria where each criterion consists of a set of operation and invariant vertices. Each slicing criterion is then used to generate a new dependency graph that represents a model fragment.

For example, for each vertex, *v*, in *ARClsAttrVSet*, Line 5 of Algorithm 7 computes a collection *Col*, where each member of *Col* is a set of operation and invariant vertices that directly depend on *v*. *ARClsAttrVSet* (see Algorithm 5) is a set of class and attributes vertices on which both operation and invariant vertices directly depend. For example, *ARClsAttrVSet* for the graph shown in Fig. 4.9b is {*MaxRoles, User, Location,*

Algorithm 7: Dependency Graph Decomposition Algorithm

```
1: Input: A dependency graph, DG, produced from the original graph by Algorithm 6
2: Output: A set of dependency graphs
3: Algorithm Steps:
4: Recompute ARClsAttrVSet for DG using Algorithm 5;
5: Compute a collection Col = S1, S2,...,Sv of operation and invariant vertex sets,
   where Sv represents a set of operation and invariant vertices that directly depend on
   vertex v, a member of ARClsAttrVSet;
6: Use the disjoint-set data structure and algorithm described in [10] to merge the
   nondisjoint-sets in Col;
7: for each set, S, in Col do
8:    Set SubVSet = {};
9:    for each vertex, V, in S do
10:      Perform a DFS from vertex V;
11:      Get a set of labeling vertices, VDFSSet, from vertex V's DFS tree;
12:      SubVSet = SubVSet ∪ VDFSSet;
13:   end for
14:   Create an empty dependency graph, SubDG;
15:   Copy all the vertices in SubVSet, from DG to SubDG;
16: end for
17: Delete DG;
```

Role$\}$. Thus *Col* for the graph is $\{\{Op6, In5\}, \{In4, Op1, Op3\}, \{In4, Op1, Op3, Op8\}, \{In4, Op1, Op3, Op8, In5\}\}$.

Line 6 uses the union-find algorithm described in [10] to merge the non-disjoint sets in *Col*, and produce a collection of sets with disjoint operation and invariant vertices. For example, *Col* for the graph shown in Fig. 4.9b becomes $\{Op6, In5, In4, Op1, Op3, Op8, In5\}$ with the union-find algorithm being used.

Lines 7-16 use each disjoint set, *S*, in *Col* to construct a new dependency graph from the input dependency graph *DG*. Lines 8-13 build a forest for *S* from each DFS tree of a vertex in *S*. Lines 14-15 create a new dependency graph that consists of all vertices in the forest. Since *Col* for the graph shown in Fig. 4.9b has only one disjoint set, the forest generated from the disjoint set consists of all the vertices in the dependency graph, indicating that the graph in Fig. 4.9b is the minimum dependency graph that cannot be

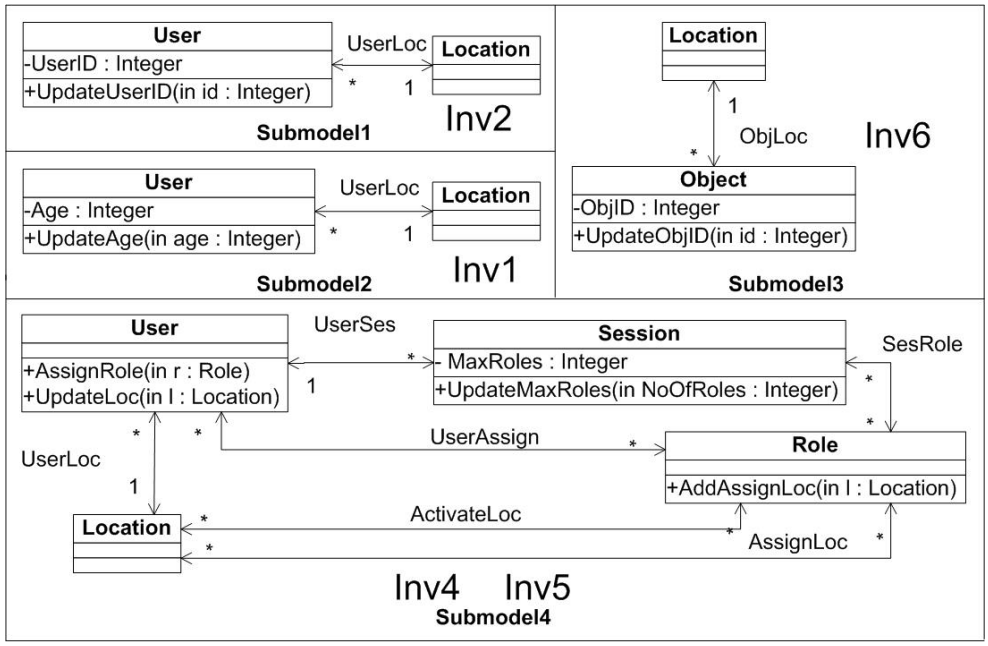


Figure 4.10: A List of Model Fragments Generated from the LRBAC Model in Fig. 4.6

decomposed further.

Figure 4.10 shows four model fragments extracted from the LRBAC model in Fig. 4.6. Each model fragment corresponds to a dependency graph in Fig. 4.9.

Chapter 5

Tool Support

We implemented a research prototype that provides (1) implementations of two proposed class model slicing techniques, and (2) a framework for evaluating the proposed slicing techniques. The prototype builds upon a number of existing technologies:

1. The prototype was developed using Java language and Eclipse development platform.

2. The UML class models used for the prototype are specified in Ecore files, the OCL invariants and operation contracts are specified in textual files, and the object configurations are specified using XMI files.

3. The prototype uses the Eclipse Modeling Framework [45] to parse Ecore class models and XMI object configurations.

4. The prototype uses the APIs from Eclipse OCL project [48] to (1) parse the OCL constraints defined in the UML class models and (2) check conformance between an object configuration and a class model with OCL invariants.

In this chapter we present the component architecture of the prototype (Section 5.1), and describe the implementations of the proposed class model slicing techniques (Section 5.2) and the evaluation framework (Section 5.3).

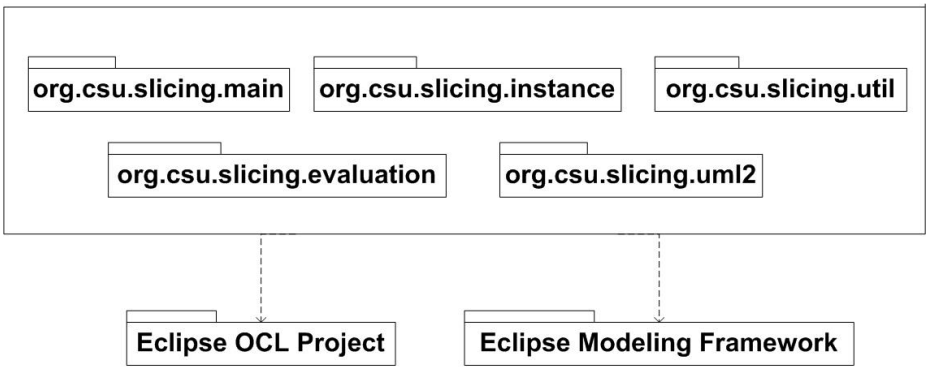


Figure 5.1: Prototype Architecture

5.1 Prototype Architecture

Figure 5.1 shows the prototype components and their usage dependencies. The *Eclipse OCL Project* and *Eclipse Modeling Framework* components are third party components. The descriptions of the non-third party components in Figure 5.1 are given below:

1. *org.csu.slicing.main*: The component acts as a driver of the model slicing framework. It provides the implementations of two proposed class model slicing techniques.

2. *org.csu.slicing.evaluation*: The component acts as a drive of the evaluation framework. It can be used to check conformance between an object configuration and a class model with OCL invariants.

3. *org.csu.slicing.instance*: The component is used to generate object configurations for the evaluation.

4. *org.csu.slicing.uml2*: The component is automatically generated by the Eclipse Modeling Framework. It provides APIs that can be used to create object configurations that conform to the UML2 class model.

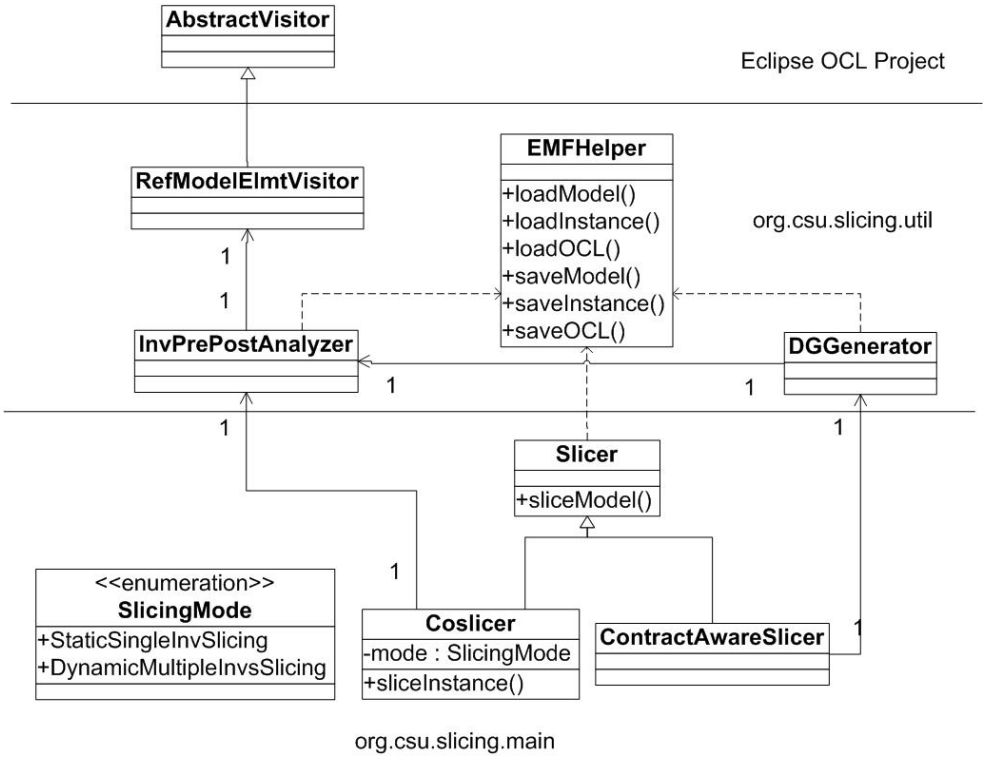


Figure 5.2: A partial UML class model that describes the implementations of the slicing techniques (part of classes, attributes, operation, and parameters are omitted)

5. *org.csu.slicing.util*: The component provides utility functions that can be used to load, prepocess and save the class models, OCL constraints, and object configurations.

5.2 Implementation of the Slicing Techniques

Figure 5.2 shows a partial UML class model that provides key classes used for the implementation of the slicing techniques. Below is the descriptions of the classes in Figure 5.2:

1. *AbstractVisitor*: The class is an abstract class from the Eclipse OCL project [48]. It provides all of the *visitXyz()* methods for the OCL metamodel (i.e., the abstract syntax tree of the OCL).

2. *RefModelElmtVisitor*: The class extends the *AbstractVisitor* class, and provides the implementations of all of the *visitXyz()* methods for the OCL metamodel. It is used to process the abstract syntax tree (AST) parsed from OCL text.

3. *InvPrePostAnalyzer*: The class loads the constraints (including invariants and operation contracts) from an OCL file and uses the *RefModelElmtVisitor* visitor to identify the model elements that are referenced by the constraints.

4. *EMFHelper*: The class provides helper functions that are used to load and save class models, OCL constraints and object configurations.

5. *DGGenerator*: The class generates a dependency graph from a class model with invariants and operation contracts.

6. *Slicer*: The class provides the implementation of the class model slicing technique. The *sliceModel()* method in the class is used to generate class model fragments.

7. *Coslicer*: The class extends the *Slicer* class, and uses the *sliceInstance()* method to generate object configuration fragments. It is used for the class model slicing technique that handles class models, object configurations and OCL invariants.

8. *ContractAwareSlicer*: The class provides the implementations of the slicing algorithms for class models with invariants and operation contracts. It is used for the class model slicing technique that handles class models with invariants and operation contracts.

5.3 Implementation of the Evaluation Framework

Figure 5.3 shows a partial UML class model that provides key classes used for the implementation of the evaluation framework. The framework provides support for two

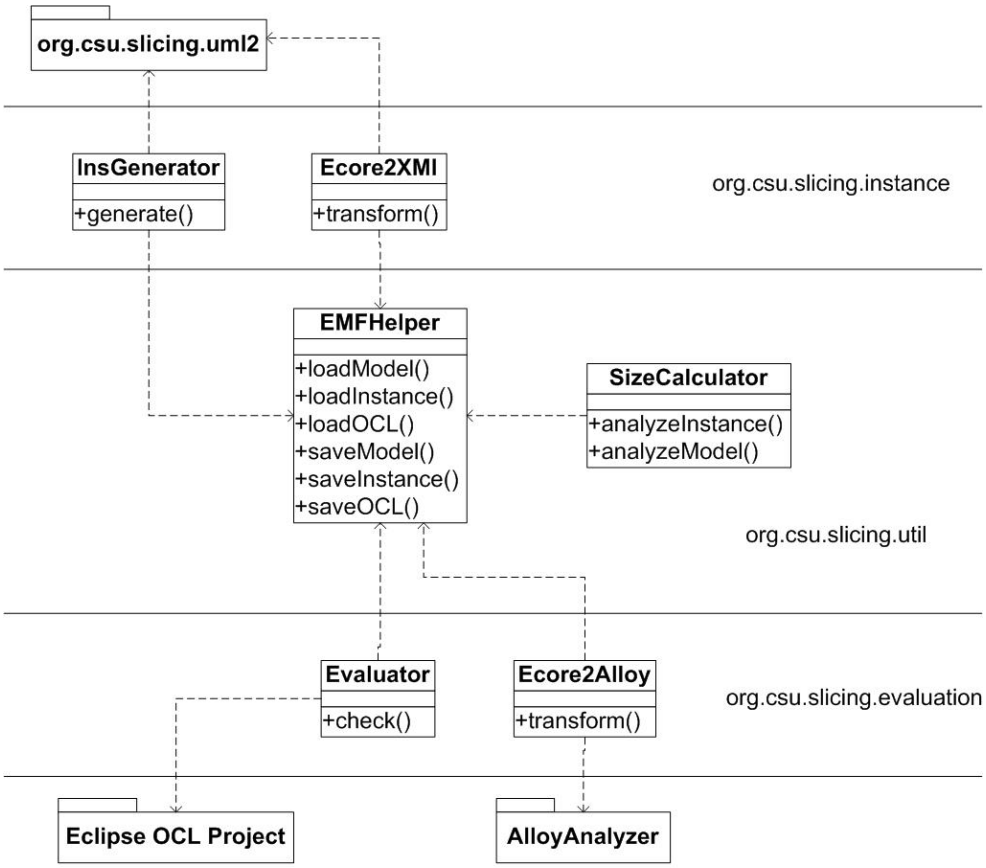


Figure 5.3: A partial UML class model that describes the implementation of the evaluation framework (part of classes, attributes, operation, and parameters are omitted)

types of class model analysis described in Chapter 3. Below is the descriptions of the classes in Figure 5.3:

1. *InsGenerator*: The class allows users to generate random object configurations (i.e., instances of the UML2 class model) used for the evaluation.

2. *Ecore2XMI*: The class converts Ecore class models to XMI object configurations (i.e., instances of the UML2 class model) used for the evaluation.

3. *SizeCalculator*: The class is used to measure the sizes of class models (e.g., number of classes) and object configurations (e.g., number of objects).

4. *Evaluator*: The class uses the OCL evaluation APIs from the Eclipse OCL project to check conformance between an object configuration and a class model with OCL invariants.

5. *Ecore2Alloy*: The class transforms Ecore class models to Alloy models. The generated Alloy models are fed into the Alloy Analyzer for rigorous analysis of invariants and operation contracts defined in the class models.

Chapter 6

Evaluation

The objective of this evaluation is to investigate the effectiveness of the proposed slicing techniques. Specifically the evaluation aims to answer the following questions:

1. *Q1*: Can the first type of slicing technique improve the efficiency of a class model analysis approach that involves checking conformance between an object configuration and a class model with specified invariants (Section 6.1)?

2. *Q2*: Can the second type of slicing technique improve the efficiency of a class model analysis approach that involves analyzing sequences of operation invocations to uncover invariant violations (Section 6.2) ?

In the remainder of this chapter we present the results of the evaluation for *Q1* (Section 6.1) and *Q2* (Section 6.2).

6.1 Evaluating Co-slicing of Class Model and Object Configuration

The effectiveness of the first type of slicing technique is determined by the speedup achieved for the analysis time. The speedup is calculated using the formula below:

$$\textit{Analysis Time Speedup (ATS)} = \frac{AT}{SAT + ST},$$

Table 6.1: A list of OCL invariants used in the evaluation

Inv1	onlyBinaryAssociationCanBeAggregations
Inv2	specializedAssociationsHasSameNumberOfEnds
Inv3	operationHasOnlyOneReturnParameter
Inv4	class1
Inv5	bc1
Inv6	ownedElementHasVisibility
Inv7	mustBeOwnedHasOwner
Inv8	nAryAssociationsOwnTheirEnds
Inv9	inheritedMemberIsValid
Inv10	isAbstractDefined
Inv11	derivedUnionIsDerived
Inv12	subsetRequiresDifferentName
Inv13	derivedUnionIsReadOnly
Inv14	isCompositeIsValid

where *AT* refers to the analysis time used for the entire class model and full object configuration, *SAT* refers to the analysis time used for the class model fragment and object configuration fragment, and *ST* refers to the slicing time.

In the remainder of this section we describe the data used for the evaluation, and the evaluation results.

6.1.1 Data Collection

The class model used for the evaluation is the UML2 metamodel because (1) it is fairly complex (768 model elements including classes, attributes, references, operations and enumerations), (2) it has a substantial number of OCL invariants (up to 53 invariants, refer to the UML2 class model in the USE tool [15]), and (3) it is widely used in both academia and industry, and thus its instances are relatively easy to collect. Table 6.1 shows a list of OCL invariants defined in the UML2 metamodel. Only 14 out of 53 invariants were used in the evaluation because the other 39 invariants specify constraints on other types of UML models (e.g., sequence diagrams) rather than class models.

The instances of the UML2 metamodel were collected from the state-of-the-art model repositories: *ReMoDD* [49] and *Metamodel Zoos* [47]. Below is a list of UML2 instances used in the evaluation:

- FSM: The class model, from the *Metamodel Zoos* repository [47], describes the concepts of a finite state machine;
- ER2MOF: The class model, from the *ReMoDD* repository [49], describes a model transformation from an entity-relationship schema to a relational model;
- HTML: The class model, from the *Metamodel Zoos* repository [47], describes the HyperText Markup Language (HTML) [33];
- MATLAB: The class model, from the *Metamodel Zoos* repository [47], describes the MATLAB language [17];
- MARTE: The class model, from the *Metamodel Zoos* repository [47], describes the MARTE profile [29].

Since the collected instances are stored in the Ecore files, we used the *Ecore2XMI* class (see Section 5.3) to generate XMI files from Ecore files. We also used the *InsGenerator* class (see Section 5.3) to generate object configurations that conform to UML2 metamodel. The instance generator ensures that at least one instance is generated for each non-abstract class of the UML2 metamodel. Three additional object configurations, namely DataSet1 (5.29 MB), DataSet2 (10.6 MB), and DataSet3 (32 MB), were used in the evaluation. Table 6.2 shows the size of the object configurations (in terms of total number of objects, slots and links) used in the evaluation.

6.1.2 Evaluation Results

Table 6.3, Table 6.4, Table 6.5, Table 6.6, Table 6.7, and Table 6.8 show the results of an evaluation we performed on a laptop computer with 2.17 GHz Intel Dual Core CPU,

Table 6.2: Size of the object configurations used in the evaluation

FSM	ER2MOF	HTML	MATLAB	MARTE	DataSet1	DataSet2	DataSet3
163	779	853	1278	7558	193778	386129	1158184

Table 6.3: Analysis time (in milliseconds) used for checking a full object configuration against a single invariant (rows 2-15) and multiple invariants (last row) respectively

	FSM	ER2MOF	HTML	MATLAB	MARTE	DataSet1	DataSet2	DataSet3
Inv1	49	69	51	80	170	686	1671	5996
Inv2	51	75	51	60	143	629	608	2122
Inv3	14	50	14	15	97	412	374	617
Inv4	44	47	69	69	85	1294	2550	3553
Inv5	49	44	74	62	87	780	671	3658
Inv6	47	51	46	47	64	499	1669	3952
Inv7	56	54	107	107	407	1807	2282	9353
Inv8	51	57	47	72	82	336	610	1560
Inv9	60	56	95	86	185	3401	4446	8549
Inv10	46	49	71	71	117	1061	1719	3560
Inv11	44	60	91	97	158	405	876	1342
Inv12	48	67	98	105	333	686	2649	3226
Inv13	45	74	86	98	188	187	281	686
Inv14	47	63	101	109	236	350	374	674
Invs	161	345	502	762	850	6122	7628	15543

3 GB RAM and Windows 7.

Table 6.3 shows the analysis time (in milliseconds) used for the entire class model (i.e., UML2) and the full object configurations (e.g., FSM) w.r.t. a single invariant (e.g., *onlyBinaryAssociationCanBeAggregations*) and all the invariants given in Table 6.1. The first row of the table gives a list of UML2 instances used in the evaluation, and the first column shows a list of invariants being checked by the evaluator described in Section 5.3. For example, the table shows that it took 49 milliseconds to check conformance between the FSM instance and the UML2 class model with the *onlyBinaryAssociationCanBeAggregations* invariant. Note that the last row Invs records the time used for checking the object configurations against all the invariants given in Table 6.1.

Table 6.4 shows the time (in milliseconds) for slicing the class model and object

Table 6.4: Time (in milliseconds) used for slicing the class model and object configurations w.r.t. a single invariant (rows 2-15) and multiple invariants (last row) respectively

	FSM	ER2MOF	HTML	MATLAB	MARTE	DataSet1	DataSet2	DataSet3
Inv1	6	15	5	31	14	157	68	192
Inv2	9	12	17	17	28	79	84	289
Inv3	2	6	6	5	15	47	50	77
Inv4	11	6	7	7	11	48	121	246
Inv5	10	5	7	5	4	30	65	187
Inv6	23	19	29	23	23	188	610	1348
Inv7	29	21	28	23	19	188	755	1245
Inv8	6	21	3	4	4	16	74	168
Inv9	17	18	20	25	26	655	1625	2814
Inv10	3	4	6	7	7	96	90	380
Inv11	3	3	5	4	4	64	28	82
Inv12	17	18	23	26	18	141	339	1283
Inv13	5	3	5	5	4	16	43	88
Inv14	3	4	3	5	5	16	50	79
Invs	59	61	71	130	117	874	1808	5321

configurations using the static, single invariant slicing technique (see rows 2-15) and the dynamic, multiple invariants slicing technique (see row 16). For example, the table shows that it took 6 milliseconds to generate a class model and an object configuration fragment from the UML2 class model and the FSM instance w.r.t the *onlyBinaryAssociationCanBeAggregations* invariant.

Table 6.5 shows the analysis time (in milliseconds) used for the UML2 model fragment and the object configuration fragments w.r.t. a single invariant and multiple invariants (see last row) defined in the UML2 model. For example, the table shows that it took 4 milliseconds to check conformance between the sliced FSM instance and the sliced UML2 class model with the *onlyBinaryAssociationCanBeAggregations* invariant. Note that the analysis time used for sliced FSM instance is 0 w.r.t *operationHasOnlyOneReturnParameter* since no object configuration fragment was generated from the FSM instance. This is because *operationHasOnlyOneReturnParameter* is an invariant defined in the context of class *Operation*, and there exists no operation object in the

Table 6.5: Analysis time (in milliseconds) used for checking an object configuration fragment against a single invariant (rows 2-15) and multiple invariants (last row) respectively

	FSM	ER2MOF	HTML	MATLAB	MARTE	DataSet1	DataSet2	DataSet3
Inv1	4	22	6	42	123	327	1477	499
Inv2	4	30	4	13	84	281	406	733
Inv3	0	9	0	0	31	172	265	141
Inv4	5	8	15	38	40	873	1241	373
Inv5	3	22	11	14	16	374	420	141
Inv6	3	22	11	14	16	249	553	1080
Inv7	16	17	67	45	192	1391	1139	5421
Inv8	3	34	2	14	49	246	402	1124
Inv9	24	19	35	39	123	2418	2777	4064
Inv10	8	4	41	18	58	624	1092	1782
Inv11	4	16	18	29	108	109	172	421
Inv12	9	25	36	45	248	364	1459	1013
Inv13	4	16	17	45	110	109	171	390
Inv14	5	26	27	37	155	140	219	499
Invs	69	225	360	405	658	5102	5332	9798

Table 6.6: Analysis time speedup achieved by the slicing technique

	FSM	ER2MOF	HTML	MATLAB	MARTE	DataSet1	DataSet2	DataSet3
Inv1	4.9	1.8649	4.6364	1.0959	1.2409	1.4174	1.0816	8.6773
Inv2	3.9231	1.7857	2.4286	2.0000	1.2768	1.7472	1.2408	2.0763
Inv3	7.0000	3.3333	2.3333	3.0000	2.1087	1.8813	1.1873	2.8303
Inv4	2.7500	3.3571	3.1364	1.5333	1.6667	1.4050	1.8722	5.7399
Inv5	3.7692	1.6296	4.1111	3.2632	4.3500	1.9307	1.3835	11.1524
Inv6	1.6786	2.1250	1.3939	1.7407	2.1333	1.1419	1.4351	1.6277
Inv7	1.2444	1.4211	1.1263	1.5735	1.9289	1.1444	1.2049	1.4031
Inv8	5.6667	1.0364	9.4000	4.0000	1.5472	1.2824	1.2815	1.2074
Inv9	1.4634	1.5135	1.7273	1.3438	1.2416	1.1067	1.0100	1.2429
Inv10	4.1818	6.125	1.5106	2.8400	1.8000	1.4736	1.4543	1.6466
Inv11	6.2857	3.1579	3.9565	2.9394	1.4107	2.3410	4.380	2.6680
Inv12	1.8462	1.5581	1.6610	1.4789	1.2519	1.3584	1.4733	1.4051
Inv13	5.0000	3.8947	3.9091	1.9600	1.6491	1.4960	1.3131	1.4351
Inv14	5.8750	2.1000	3.3667	2.5952	1.4750	2.2436	1.3903	1.1661
Invs	1.2578	1.2063	1.1647	1.4243	1.0968	1.0244	1.0684	1.0280

FSM instance. The analysis time speedup shown in Table 6.6 is calculated using the *AST* formula given at the beginning of this section. The speedups achieved for large in-

Table 6.7: Size of the object configuration fragments in terms of total number of object elements including objects, slots and links

	FSM	ER2MOF	HTML	MATLAB	MARTE	DataSet1	DataSet2	DataSet3
Inv1	34	173	133	271	1226	4016	8000	24000
Inv2	43	194	195	320	1624	15060	30000	90000
Inv3	0	23	0	0	38	1506	3000	9000
Inv4	6	18	59	45	324	6514	13045	39017
Inv5	6	25	59	45	334	7530	15000	45000
Inv6	38	160	241	295	1884	78312	156000	468000
Inv7	54	247	301	448	2579	89858	179000	537000
Inv8	34	173	133	271	1226	4093	8180	24503
Inv9	38	241	160	295	1884	78814	157000	471000
Inv10	15	56	73	88	515	12250	25000	75000
Inv11	16	71	112	163	998	2005	4037	12007
Inv12	64	284	364	498	2994	78814	157000	471000
Inv13	16	71	112	163	998	1747	3527	10518
Inv14	16	71	112	163	998	1753	3481	10468
Invs	132	638	686	1050	5844	97668	194656	583873

stances (1.0244 for DataSet1, 1.0684 for DataSet2, 1.0280 for DataSet3) w.r.t. multiple invariants are low since the slicing time for these instances is relatively high. This is mainly because dynamic multiple invariants slicing may remove less elements from the analysis inputs.

Table 6.7 shows the size of sliced instances w.r.t each invariant and all the invariants given in Table 6.1. Given that the full instances have 163 (FSM), 779 (ER2MOF), 853 (HTML), 1278 (MATLAB), 7558 (MARTE), 193778 (DataSet1), 386129 (DataSet2), and 1158184 (DataSet3) object elements respectively (see Table 6.2), the slicing technique significantly reduced the size of instances being analyzed. The slicing technique also reduced the size of the UML2 model being analyzed. Table 6.8 shows the size of the generated UML2 model fragments w.r.t each invariant and all the invariant given in Table 6.1. For example, the UML2 model fragment that is referenced by the *onlyBinaryAssociationCanBeAggregations* invariant has only 10 model elements while the entire UML2 model has 768 model elements. For multiple invariants slicing, the generated

Table 6.8: Size of the UML2 model fragments w.r.t each invariant and all the invariants given in Table 6.1

Inv1	Inv2	Inv3	Inv4	Inv5	Inv6	Inv7	Inv8	Inv9	Inv10	Inv11	Inv12	Inv13	Inv14	Invs
10	39	5	15	21	200	228	9	204	33	5	199	6	6	291

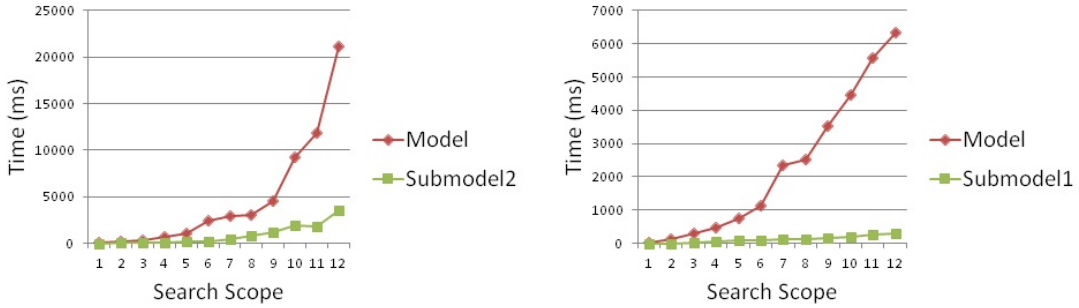
UML2 model fragment has 291 model elements.

We also applied the slicing technique to a variety of tools (e.g., Kermeta Workbench [12][21], USE [15] and Alloy Analyzer [18]), showing that the slicing technique is tool independent.

6.2 Evaluating Contract-aware Slicing of Class Model

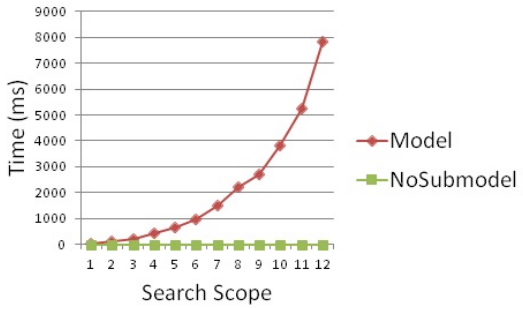
The *AST* formula given at the beginning of Section 6.1 is used to calculate the analysis speedup achieved by the first type of slicing technique that produces only one model fragment from a class model. The formula depends on the slicing time and the analysis time used for one class model and one model fragment. Unlike the first type of slicing technique, the second type of slicing technique may produce more than one model fragments from a class model. Thus the *AST* formula cannot be used to estimate the effectiveness of the second type of slicing technique. In addition, the second type of class model analysis approach requires users to specify the search scope (i.e., the maximum number of instances the Alloy Analyzer can produce for a class) for each class model and model fragment. Therefore, we measure the effectiveness of the second type of slicing technique by comparing the analysis time used for each invariant in the entire class model and the corresponding model fragment respectively w.r.t. the Alloy search scope. The evaluation was conducted on a laptop computer with 2.17 GHz Intel Dual Core CPU, 3 GB RAM, Windows 7 and the Alloy Analyzer (version 4.2 with SAT4J).

In the remainder of this section we describe the evaluation results for the LRBAC class model (Section 6.2.1) and other class models (Section 6.2.2).

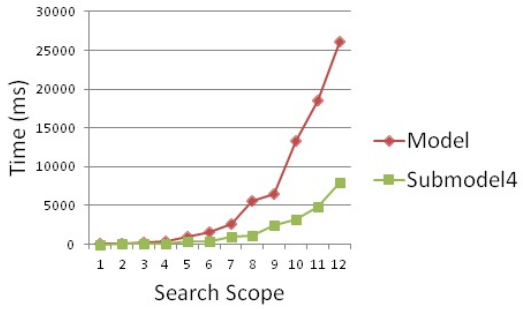


(a) Analyzing Inv1

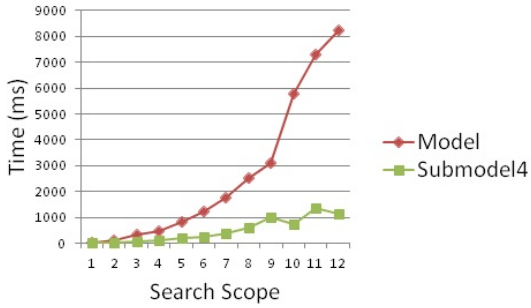
(b) Analyzing Inv2



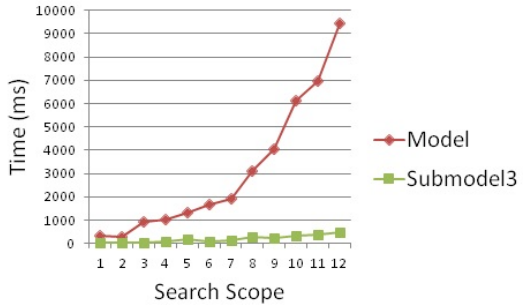
(c) Analyzing Inv3



(d) Analyzing Inv4



(e) Analyzing Inv5



(f) Analyzing Inv6

Figure 6.1: Analyzing the Invariants given in Table 4.1 in the Entire LRBAC Model and the Corresponding Model Fragments

6.2.1 Evaluation Results for LRBAC

We applied the slicing technique to the LRBAC class model in Fig. 4.6, and it took 827 milliseconds to decompose the class model into 4 fragments as shown in Figure Fig. 4.10. The Alloy Analyzer was used to analyze the invariants given in Table 4.1

with the class model and the corresponding model fragment respectively.

Figure 6.1 shows the results of an evaluation we performed on the entire LRBAC class model and the corresponding model fragments. Each subfigure in 6.1 has an *x* axis, namely *SearchScope*, indicating the maximum number of instances the Alloy Analyzer can produce for a class, and a *y* axis, namely *Time*, showing the total analysis time (in millisecond) for building the SAT formula and finding an Alloy instance. For example, Fig. 6.1a shows the time used to analyze the invariant *Inv*1 in the entire model *Model* and the model fragment *Submodel*2 respectively.

The difference between the time used for analyzing *Model* and that for *Submodel*2 is relatively small when the Alloy search scope is below 5. For a search scope above 10, the time used for analyzing *Model* becomes significantly large while that for *Submodel*2 is still below 5000 ms. Fig. 6.1c shows that the analysis time for the *Inv*3 invariant remains at 0, and *Inv*3 is not included in any model fragment. This is because *Inv*3 was removed after the dependency analysis identified the invariant as an analysis-irrelevant element.

Note that the four algorithms described in the paper use set addition/deletion operations, a depth-first-search algorithm, and a disjoint-set algorithm. Thus the execution time for implementations of these four algorithms should not increase significantly as the size of the class model increases. Since the execution time of SAT solver-based tools (e.g., Alloy) could be exponential on the size of the class model, the slicing algorithm described in the paper could speed up the verification process for large models.

The evaluation also showed that the analysis results are preserved by the slicing technique. For example, the analysis performed on the unsliced model and the model fragments both found that the constraints specified in invariant *Inv*2, *Inv*5 and *Inv*6 were violated by operation *UpdateUserID*, *UpdateMaxRoles* and *UpdateObjID* respectively. The analysis of the full model also revealed that Inv3 is not violated by operation invoca-

Table 6.9: Size of the class models and the number of OCL invariants and operation contracts defined in the models

Model	Size	# of Invs	# of Op Contracts
CarRental	48	8	2
Project	47	3	4
CoachBus	66	6	6
RoyalAndLoyal	151	9	6

tions; this is consistent with the identification of Inv3 as an analysis-irrelevant element.

6.2.2 Evaluation Results for Other Class Models

Below is a list of class models used in the evaluation:

- *CarRental*: The class model, from the USE project [15], describes the concepts of the car rental service;
- *Project*: The class model, from the USE Project [15], describes the concepts of a project management system;
- *CoachBus*: The class model, from the class model slicing paper [40], describes the information system of a bus company;
- *RoyalAndLoyal*: The class model, from the OCL book [50], describes loyalty programs for companies that offer their customers various kinds of bonuses.

Table 6.9 shows the size of these class models (in terms of total number of class model elements including classes, attributes, references, operations, and enumerations) and the number of invariants and operation contracts defined in the models. For example, the CarRental class model has 48 elements, 8 invariants and 2 operation contracts. Note that the CoachBus class model used in the evaluation is not the same as the one used in Section 4.1 because it has 6 operation contracts.

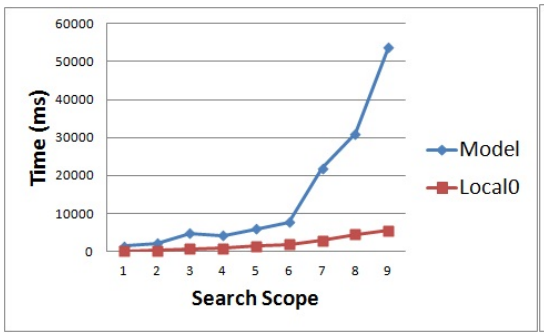
Table 6.10: Slicing Results

Model	Slicing Time (ms)	Fragment	Size	# of Invs	# of Op Contracts
CarRental	889	Local0 Leftover1	4 9	1 1	1 1
Project	846	Leftover0	24	3	4
CoachBus	780	Local0 Local1 Leftover2	4 4 12	1 1 2	1 1 3
RoyalAndLoyal	837	Locl0 Loftover1	6 32	1 7	1 2

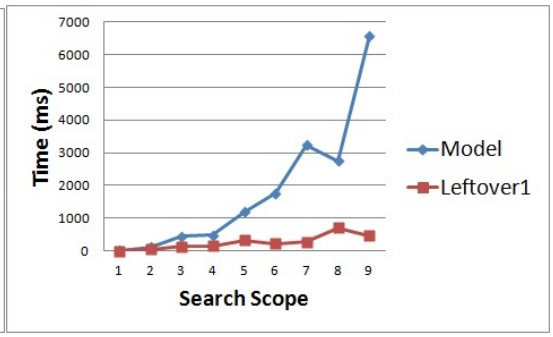
We applied the slicing technique to these models, and Table 6.10 shows the slicing results. For example, it took 889 milliseconds to generate two fragments, namely Local0 and Leftover1, from the CarRental class model. Local0 has 4 elements, 1 invariant and 1 operation contract, and Leftover1 has 9 elements, 1 invariant and 1 operation contract.

The Alloy Analyzer was used to analyze the invariants defined in the class model. Specifically for an invariant defined in a class model, the Alloy Analyzer checks the invariant with the class model and the corresponding model fragment respectively. Figure 6.2 shows the results of an analysis we performed on the entire CarRental class model and the corresponding model fragments. Two fragments, namely Local0 and Leftover1 (see Table 6.10), were generated from the CarRental class model.

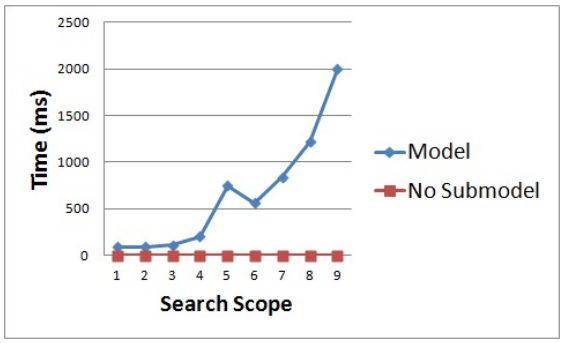
Both the time used for analyzing Person3 in Local0 and the time used for analyzing Brach2 in Leftover1 are relatively small even when the Alloy search scope is up to 9. Note that there are 9 invariants in the CarRental class model, while only 2 invariants exists in the generated model fragments. This is because the rest of invariants were removed after the dependency analysis identified them as analysis-irrelevant elements. For example, Fig. 6.2c shows that the invariant Employee1 is not included in any model fragment, and its analysis time remains at 0, which indicated that the invariant will not be violated by any operation invocation. The analysis of the entire model revealed that



(a) Analyzing Invariant Person3



(b) Analyzing Invariant Branch2

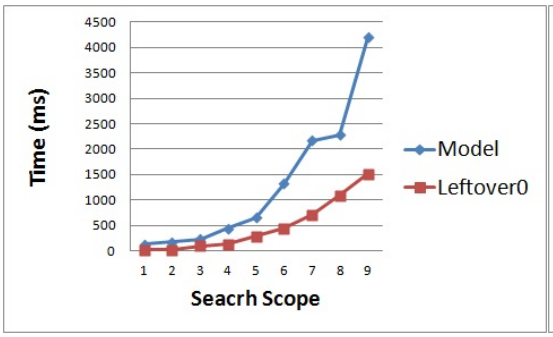


(c) Analyzing Invariant Employee1

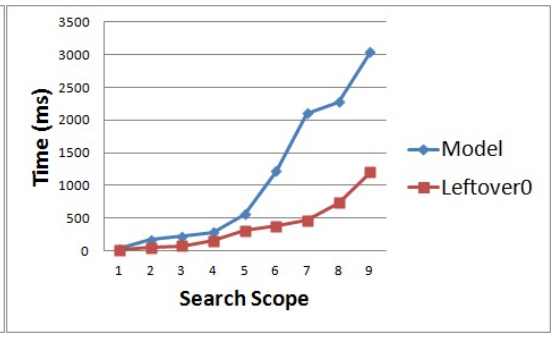
Figure 6.2: Analyzing the Invariants in the Entire CarRental Class Model and the Corresponding Model Fragments

Employee1 is not violated by operation invocations; this is consistent with the identification of Employee1 as an analysis-irrelevant element.

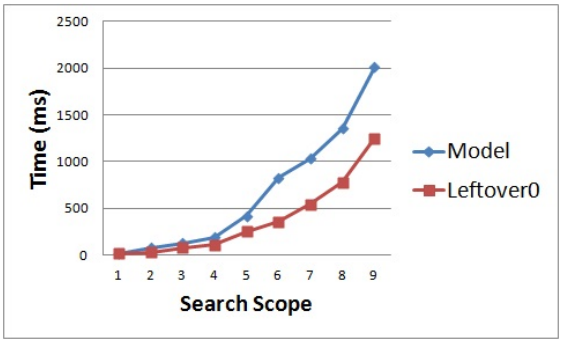
Figure 6.3 shows the results of an analysis we performed on the entire Project class model and the corresponding model fragments. In this case, only one model fragment, Leftover0, was generated from the Project class model (see Table 6.10), and the fragment share the same invariants and operation contracts with the class model. Since the model fragment has fewer elements than the entire class model, it could be expected that the analysis time used for the model fragment is smaller than that for the class model. This is confirmed by the analysis results given in Figure 6.3. However, since the number of elements that were removed from the class model is not quite large, the difference between the analysis time used for the model fragment and that for the class model is



(a) Analyzing Invariant OnlyOwnEmployeesInProject



(b) Analyzing Invariant NotOverloaded

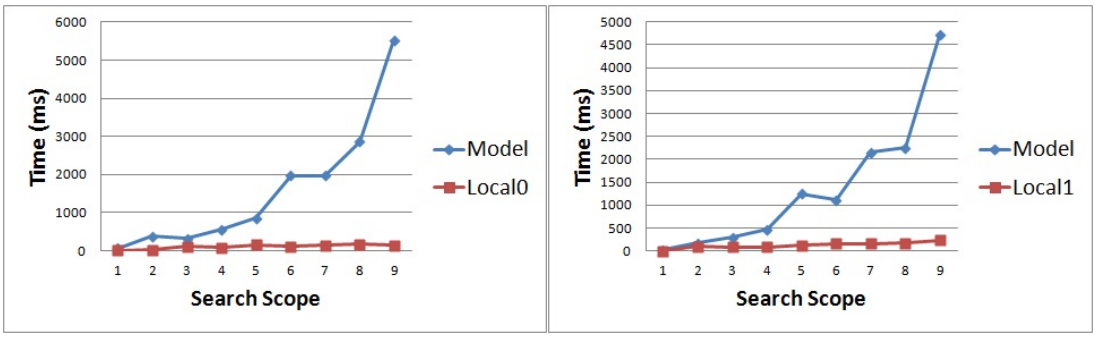


(c) Analyzing Invariant AllQualificationsForActiveProject

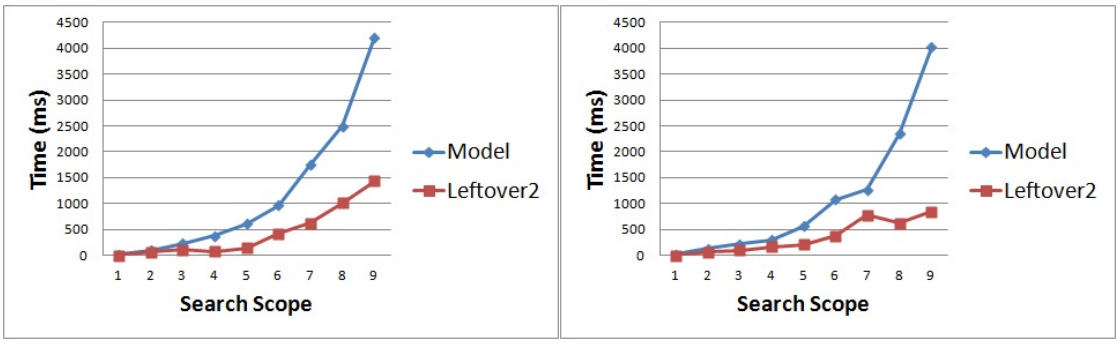
Figure 6.3: Analyzing the Invariants in the Entire Project Class Model and the Corresponding Model Fragments

not significant when the Alloy search scope is small.

Figure 6.4 shows the results of an analysis we performed on the entire CoachBus class model and the corresponding model fragments. Three fragments, namely Loca0, Local1 and Leftover2 (see Table 6.10), were generated from the CoachBus class model. Both the time used for analyzing NonNegativeAge in Local0 and the time used for analyzing UniqueTicketNumber in Local1 are relatively small (less than 300 milliseconds) even when the Alloy search scope is up to 9. The time used for analyzing MinCoachSize in Leftover2 steadily grows as the Alloy search scope increases. But it is still less than that for analyzing MinCoachSize in the entire CoachBus class model. Note that the CoachBus class model has 6 invariants (see Table 6.9), and there are 2 invariants from



(a) Analyzing Invariant NonNegativeAge (b) Analyzing Invariant UniqueTicketNumber



(c) Analyzing Invariant MinCoachSize (d) Analyzing Invariant MaxCoachSize

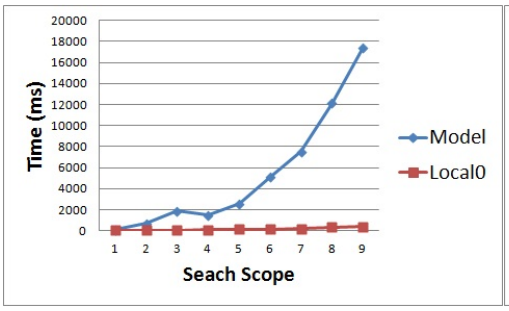
Figure 6.4: Analyzing the Invariants in the Entire CoachBus Class Model and the Corresponding Model Fragments

the CoachBus class model that are not shown in the analysis results because they were removed as analysis-irrelevant elements.

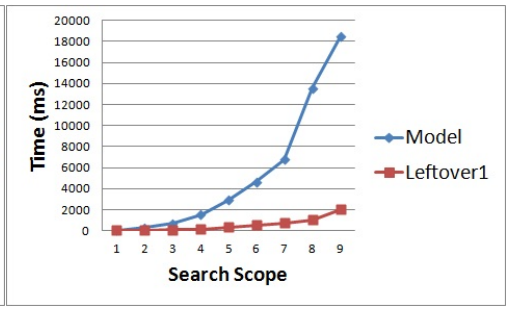
Figure 6.5 shows the results of an analysis we performed on the entire RoyalAndLoyal class model and the corresponding model fragments. Two fragments, namely Local0 and Leftover1 (see Table 6.10), were generated from the RoyalAndLoyal class model. The Local0 fragment has only 1 invariant (UniqueName) and 1 operation contract. Compared with the time used for analyzing UniqueName in the entire class model, the time used for analyzing UniqueName in Local0 is significantly small (less than 500 milliseconds). The time used for analyzing ProgramPartner1, nrOfParticipants, ServiceLevel1, Customer10, sizesAgree CustomerCard3 in Leftover1 is less than 2000 milliseconds) even when the Alloy search scope is up to 9. The time used for ana-

lyzing Customer1 in Leftover1 steadily grows as the Alloy search scope increases, and it reaches 4000 milliseconds when the search scope is up to 9. At this point, the time used for analyzing Customer1 in the entire class model is up to 24000 milliseconds. Note that the RoyalAndLoyal class model has 9 invariants (see Table 6.9), there is 1 invariant from the RoyalAndLoyal class model that is not shown in the analysis results because it was removed as analysis-irrelevant element.

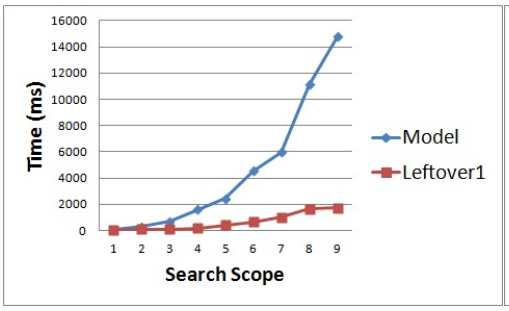
All the evaluation results show that the slicing technique did improve the efficiency of the class model analysis. The evaluation also showed that the analysis results are preserved by the slicing technique.



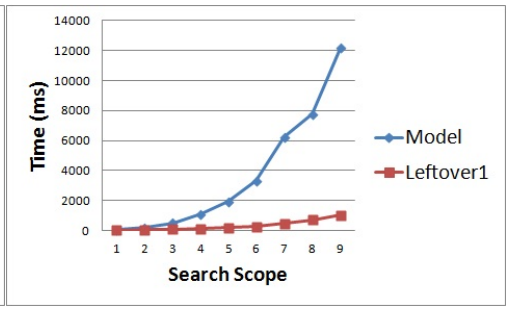
(a) Analyzing Invariant UniqueName



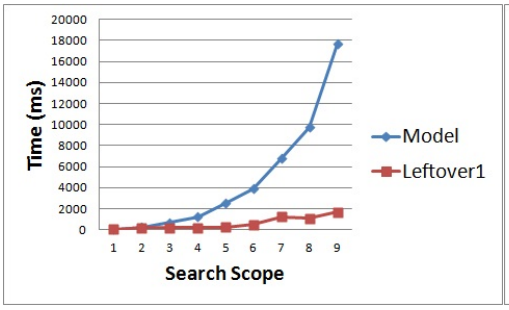
(b) Analyzing Invariant ProgramPartner1



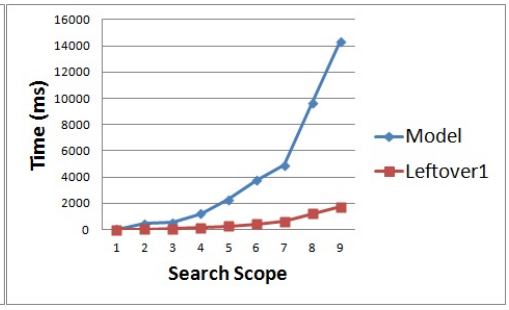
(c) Analyzing Invariant nrOfParticipants



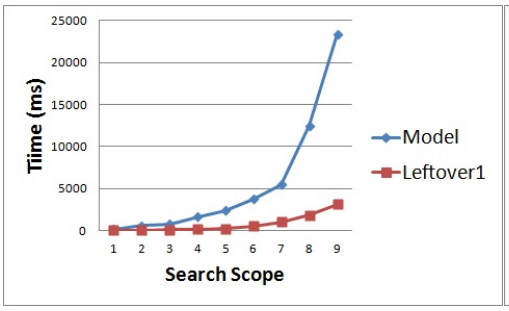
(d) Analyzing Invariant ServiceLevel1



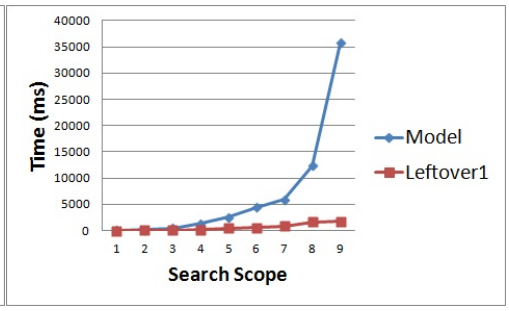
(e) Analyzing Invariant Customer10



(f) Analyzing Invariant sizesAgree



(g) Analyzing Invariant Customer1



(h) Analyzing Invariant CustomerCard3

Figure 6.5: Analyzing the Invariants in the Entire RoyalAndLoyal Class Model and the Corresponding Model Fragments

Chapter 7

Conclusion

In this chapter we summarize contritions of the dissertation (Section 7.1), discuss limitations of the proposed slicing techniques (Section 7.2), and present an overview of plans for future related work (Section 7.3).

7.1 Contribution

We presented a slicing technique for improving the efficiency of class model analysis that involves checking if an instance of a class model satisfies the invariants defined in the class model. The slicing technique produces a fragment from a class model with invariants and an object configuration. The fragment can be more efficiently analyzed by conforming checking tools.

We also presented a slicing approach for UML class models that includes OCL invariants and operation contracts. The slicing approach 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. The approach takes as input a UML class model with operation contracts and OCL invariants, and decomposes the model into model fragments with disjoint operations and invariants.

We developed a platform that provides implementations of two proposed class model slicing techniques, and an evaluation framework for the proposed slicing techniques.

The results of the evaluation we performed showed that the proposed slicing techniques can significantly reduce the time to perform the analysis.

7.2 Discussion

The slicing technique on class model and object configuration is limited in its ability to produce a smaller fragment from a larger class model and object configuration w.r.t. the slicing criterion (i.e., an invariant defined the class model). There may be invariants that reference all elements of a class model. In this case the slicing technique cannot ensure that a smaller fragment will be produced from a class model and object configuration. An empirical study conducted by Juan et al. [9] showed that in practice a substantial number of invariants only reference part of the class model in which they are defined. Thus our slicing technique is expected to work for many class models used in practice.

The slicing technique on class model with invariants and operation contracts is limited in its ability to produce smaller model fragments from a large class model w.r.t. the OCL constraints (i.e., invariants and operation contracts). There may be constraints that reference all model elements of a class model and thus require the entire class model to be present when analyzed (reflecting a very tight coupling across all model elements). In this case, the slicing technique described in this proposal does not ensure that more than one independently analyzable fragments will be produced from a class model.

7.3 Future Work

One objective of our future work is to address the limitation of our slicing techniques. We plan to use class model refactoring techniques to reduce the coupling across elements of a class model if the proposed slicing techniques cannot produce smaller fragments from the class model.

Anther future direction of this work could be slicing invariants using the informa-

tion found in object configurations. For example, a complex invariant may reference a substantial number of class model elements, while an object configuration may only reference a small subset of class model elements. In this case, checking the object configuration against the invariant would not require the entire invariant to be analyzed. Thus the slicing technique would use the information in the object configuration to reduce the complex invariant into smaller subinvariants, where only a subset of the subinvariants are needed to analyze the object configuration.

We also plan to use a variety of OCL constraints as slicing criteria in the evaluation. Specifically we are currently investigating how we can use OCL benchmark given in [16] to generate complex OCL constraints.

REFERENCES

[1] K. Anastasakis. UML2Alloy Reference Manual. *UML2Alloy Version: 0.52 [Online] available at http://www. cs. bham. ac. uk/~ bxb/UML2Alloy/files/uml2alloy_manual. pdf (retrieved 01/09/2009)*, 2012.

[2] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A Challenging Model Transformation. In *Model Driven Engineering Languages and Systems*, pages 436–450. Springer, 2007.

[3] 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 33rd International Conference on Software Engineering (ICSE)*, pages 291–300. IEEE, 2011.

[4] J. Bae and H. Chae. UMLSlicer: A tool for modularizing the UML metamodel using slicing. In *Proceedings of the 8th IEEE International Conference on Computer and Information Technology*, pages 772–777. IEEE, 2008.

[5] J. Bae, K. Lee, and H. Chae. Modularization of the UML metamodel using model slicing. In *Fifth International Conference on Information Technology: New Generations*, pages 1253–1254. IEEE, 2008.

[6] A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux. Modeling model slicers. *In Model Driven Engineering Languages and Systems*, pages 62–76, 2011.

[7] A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux. Kompren: modeling and generating model slicers. *Software & Systems Modeling*, pages 1–17, 2012.

[8] B. Bordbar and K. Anastasakis. UML2ALLOY: A tool for lightweight modelling of discrete event systems. In *IADIS AC*, pages 209–216, 2005.

[9] J. Cadavid, B. Combemale, and B. Baudry. Ten years of Meta-Object Facility: an analysis of metamodeling practices. In *Technical Report by the Triskell Team at INRIA/IRISA*, pages 1–25, 2012.

[10] T.H. Cormen. *Introduction to algorithms*. The MIT press, 2001.

[11] R. Eshuis and R. Wieringa. Tool support for verifying UML activity diagrams. *IEEE Transactions on Software Engineering*, 30(7):437–447, 2004.

[12] F. Fleurey, Z. Drey, D. Vojtisek, C. Faucher, and V. Mahé. Kermeta language, reference manual. *Internet: http://www. kermeta. org/docs/KerMeta-Manual. pdf. IRISA*, 2006.

[13] R. France, A. Evans, K. Lano, and B. Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, 1998.

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

[15] M. Gogolla, F. Büttner, and M. Richters. USE: A uml-based specification environment for validating UML and OCL. *Science of Computer Programming*, 69(1):27–34, 2007.

[16] M. Gogolla, M. Kuhlmann, and F. Buttner. A benchmark for OCL engine accuracy, determinateness, and efficiency. In *Model Driven Engineering Languages and Systems*, pages 446–459. Springer, 2008.

[17] MATLAB Users Guide. The mathworks inc. *Natick, MA*, 4:382, 1998.

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

[19] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: The alloy constraint analyzer. In *Proceedings of the 22th International Conference on Software Engineering*, pages 730–733. IEEE, 2000.

[20] C. Jeanneret, M. Glinz, and B. Baudry. Estimating footprints of model operations. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 601–610. IEEE, 2011.

[21] J.M. Jézéquel, O. Barais, and F. Fleurey. Model driven language engineering with kermeta. *Generative and Transformational Techniques in Software Engineering III*, pages 201–221, 2011.

[22] 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)*, pages 635–638. Ieee, 2005.

[23] S. Keele. Guidelines for performing systematic literature reviews in software engineering. Technical report, EBSE Technical Report EBSE-2007-01, 2007.

[24] B. Korel, I. Singh, L. Tahat, and B. Vaysburg. Slicing of state-based models. In *Proceedings of International Conference on Software Maintenance (ICSM)*, pages 34–43. IEEE, 2003.

[25] J. Lallchandani and R. Mall. Slicing UML architectural models. *ACM SIGSOFT Software Engineering Notes*, 33(3):4, 2008.

[26] J. Lallchandani and R. Mall. A dynamic slicing technique for UML architectural models. *IEEE Transactions on Software Engineering*, 37(6):737–771, 2011.

[27] K. Lano and S. Kolahdouz-Rahimi. Slicing of UML models using model transformations. *In Model Driven Engineering Languages and Systems*, pages 228–242, 2010.

[28] K. Lano and S. Kolahdouz-Rahimi. Slicing techniques for UML models. *Journal of Object Technology*, 10, 2011.

[29] Frédéric Mallet and Robert De Simone. Marte: a profile for rt/e systems modeling, analysis–and simulation? In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, page 43. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[30] Bertrand Meyer. Applying'design by contract'. *Computer*, 25(10):40–51, 1992.

[31] A. Muller, F. Fleurey, and J. Jézéquel. Weaving executability into object-oriented meta-languages. In *Model Driven Engineering Languages and Systems*, pages 264–278. Springer, 2005.

[32] QVT Omg. Meta Object Facility (MOF) 2.0 query/view/transformation specification. *Final Adopted Specification (November 2005)*, 2008.

[33] Dave Raggett, Arnaud Le Hors, Ian Jacobs, et al. Html 4.01 specification. *W3C recommendation*, 24, 1999.

[34] I. Ray and M. Kumar. Towards a location-based mandatory access control model. *Computers & Security*, 25(1):36–44, 2006.

[35] I. Ray, M. Kumar, and L. Yu. LRBAC: A location-aware role-based access control model. *Information Systems Security*, pages 147–161, 2006.

[36] I. Ray and L. Yu. Short paper: Towards a location-aware role-based access control model. In *Proceedings of the First International Conference on Security and Privacy for Emerging Areas in Communications Networks*, pages 234–236. IEEE, 2005.

[37] M. Richters and M. Gogolla. Validating UML models and OCL constraints. In *the UML 2000 Unified Modeling Language*, pages 265–277. Springer, 2000.

[38] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[39] S. Sen, N. Moha, B. Baudry, and J. Jézéquel. Meta-model pruning. In *Model Driven Engineering Languages and Systems*, pages 32–46. Springer, 2009.

[40] 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*, pages 185–194. ACM, 2010.

[41] A. Shaikh and U. Wiil. UMLtoCSP (UOST): a tool for efficient verification of UML/OCL class diagrams through model slicing. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 37. ACM, 2012.

[42] A. Shaikh, U. Wiil, and N. Memon. Uost: UML/OCL aggressive slicing technique for efficient verification of models. In *System Analysis and Modeling: About Models*, pages 173–192. Springer, 2011.

[43] A. Shaikh, U.K. Wiil, and N. Memon. Evaluation of tools and slicing techniques for efficient verification of UML/OCL class diagrams. *Advances in Software Engineering*, 2011, 2011.

[44] O.M.G.A. Specification. Object constraint language, 2007.

[45] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2008.

[46] W. Sun, R. France, and I. Ray. Rigorous analysis of UML access control policy models. In *Proceedings of IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY)*, pages 9–16. IEEE, 2011.

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

[48] Eclipse OCL Project Team. Eclipse OCL Project. In *http://projects.eclipse.org/projects/modeling.mdt.ocl*. Eclipse Community, 2005.

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

[50] Jos B Warmer and Anneke G Kleppe. *The object constraint language: getting your models ready for MDA*. Addison-Wesley Professional, 2003.

[51] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Press, 1981.

[52] L. Yu, B. France, I. Ray, and W. Sun. Systematic scenario-based analysis of UML design class models. In *Proceedings of 17th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 86–95. IEEE, 2012.

[53] L. Yu, R. France, and I. Ray. Scenario-based static analysis of UML class models. *In Model Driven Engineering Languages and Systems*, pages 234–248, 2008.

[54] L. Yu, R. France, I. Ray, and S. Ghosh. A Rigorous Approach to Uncovering Security Policy Violations in UML Designs. In *Proceedings of 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 126–135. IEEE, 2009.

[55] L. Yu, R. France, I. Ray, and K. Lano. A light-weight static approach to analyzing UML behavioral properties. In *Proceedings of 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS)*, pages 56–63, 2007.

[56] T. Yue. *Ph.D. Dissertation: Automatically deriving a UML analysis model from a use case model*. Carleton University, 2010.