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

Spring 2015

Abstract

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.

Acknowledgements

I am truly grateful to many people for this dissertation. I would not have been able to complete the dissertation without their support and help.

I owe special thanks to my advisor, Dr. Robert B. France for his untiring support, inspiration and guidance. He was the person who has been a tremendous source of inspiration to this dissertation. He spent countless hours to help me make my pursuit of the Ph.D. a successful and fulfilling journey. Even though he is no longer with us, he will be always living in our minds.

I would like to express my sincere thanks to my advisor, Dr. Indrakshi Ray for her untiring support, encouragement and help. Her insightful comments and feedback helped me to refine my research work. I would like to thank my Ph.D. committee members Dr. James M. Bieman, Dr. Yashwant K. Malaiya and Dr. Daniel S. Cooley for their valuable advice and help.

I am also grateful to the faculty of the computer science department, especially Dr. Sudipto Ghosh, Dr. A. P. Willem Bohm, Dr. Sanjay Rajopadhye, Dr. Indrajit Ray, Dr. Chris Wilcox and Mr. Russ Wakefield for their help and friendship. I would like to thank the staff of the computer science department, especially Kim and Sharon, for granting me a favor all the time. I would like to extend many thanks to my friends and fellow graduate students at CSU.

Last but not the least, I would like to express my deepest gratitude to my dear father, who passed away in August 2009, for his love, encouragement and sacrifice, to my dear mother, for her love, kindness, and support, and to my dear wife, for her love and support.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

1.1. Problem

Research in Model-Driven Development (MDD) targets the rapidly growing complexity of software system. MDD research aims to produce technologies that allow system developers to raise the level of abstraction at which complex software systems are developed. This is accomplished through the use of models that can be analyzed and transformed to concrete implementations. However, design errors in the models may be propagated into the implementations via model-to-code transformation. If a design model contains errors that are not removed before transformation, those errors can be passed to the generated code where they can be more expensive to remove. Therefore, for MDD approaches to succeed, it is very important to uncover design errors in models before they are used to generate code.

Rigorous analysis of models can enhance the ability of developers to understand the system under development and to identify potentially costly design problems during the early stages of software development. Class models expressed using the Unified Modeling Language (UML) [15] 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 [17][36][19][18][14][21][31]). 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. For example, while design models were built by hand in the early days of MDD, nowadays models with possibly more than one million of elements can be built programatically (e.g., reverse engineering [6]). The scalability of current class model analysis tools will become an issue in these situations. There is thus a need for techniques that support scalable rigorous analysis of class models.

Slicing techniques [48] produce reduced forms of artifacts that can be used to support, for example, scalable analysis of artifact properties. Slicing techniques have been proposed for different software artifacts, including programs (e.g., see [16][48]), and models (e.g., see [3][7][13][22][24]). In the MDD area, model slicing techniques have been used to support a variety of modeling tasks, including model comprehension [3][7][24], analysis [20][27][28], and verification [13][39][42].

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 satisfying a *slicing criterion*) and the rest of the model is analyzed using heuristics related to a model's properties (e.g., the structure of a model); and (2) a fragment of the model consisting only of elements satisfying a slicing criterion and their dependent model elements, is extracted from the model.

For example, in the context of class model analysis, a slicing criterion involves class model elements being analyzed, and a generated class model fragment consists of class model elements being analyzed and their dependent model elements. The dependency analysis between a slicing criterion and a class model ensures that the analysis results are preserved by the slicing techniques, that is, the analysis results on the model fragment will be the same as with the entire class model. This assertion is based on the observation that the slicing technique produces the fragment by identifying the model elements that are directly referenced by the slicing criterion and analyzing their dependencies with other model elements.

Rigorous analysis of invariants and operation contracts expressed in the Object Constraint Language (OCL) [43] 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. Therefore, the existing class model slicing techniques cannot be applied to situations in which (1) invariants are used to specify additional properties that cannot be expressed directly in a class model (e.g., an acyclicity property), and (2) model-based software development approaches 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 class model analysis techniques, (1) a technique for checking conformance between an object configuration and a class model with specified invariants (i.e., invariant checking) and (2) a technique for analyzing sequences of operation invocations to uncover invariant violations.

Invariant Checking involves determining whether an instance (i.e., object model) of a class model satisfies the well-formed rules (i.e., invariants) defined in the class model. It plays an important role in MDD in that it can improve developers' understanding of complex systems and uncover structural errors in design models during the early stages of software development. In this dissertation we describe an invariant checking approach that introduces the model slicing technique to the invariant checking process. The approach aims to improve the scalability of existing invariant checking tools used to determine whether an object model satisfies the invariants defined in a class model. The approach is not intended to improve the existing invariant checking algorithms. Instead, the approach aims to reduce the size of the checking inputs to make the analysis more efficient. Therefore the approach described in the dissertation is agnostic to the technological space. It means our approach preprocesses the input of the invariant checking process, and thus is agnostic to the checking technologies the software developers are working with.

In this dissertation we focus on analysis that involves checking the consistency between an object model and the invariants defined in a class model, and checking whether an object model is a valid instance of a class model is out of scope of the paper. Thus the precondition of our approach is that the input object model must be a valid instance of the input class model. Note that a valid instance of a class model, particularly in this dissertation, refers to an object model that conforms to the structural constraints (e.g., multiplicity constraints) defined in the class model, and it may or may not satisfy the invariants defined in the class model.

We also propose a slicing technique to improve the efficiency of a model analysis technique we developed (*Contract Checking*) to check that operation contracts do not allow invariant violations when sequences of conforming operations are invoked [45]. 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.

1.3. Evaluation

We implemented a research prototype that provides implementations of two proposed class model slicing techniques. The prototype was developed using the Java language and Eclipse development platform, and builds upon the Eclipse Modeling Framework [44]. We also developed an evaluation framework to investigate the effectiveness and correctness of the proposed slicing techniques. The evaluation aims to answer the following questions:

(1) *Q1*: Can the slicing technique improve the efficiency of the invariant checking and preserve the checking results?

    Specifically the evaluation will explore whether (1) the use of the slicing technique can reduce the invariant checking time (i.e., effectiveness of the slicing technique), and (2) the checking results remain the same in the context of the slicing technique (i.e., the correctness of the slicing technique).

(2) *Q2*: Can the slicing technique improve the efficiency of the contract checking and preserve the checking results?

    Specifically the evaluation will explore whether (1) the use of the slicing technique can reduce the contract checking time (i.e., effectiveness of the slicing technique), and (2) the checking results remain the same in the context of the slicing technique (i.e., the correctness of the slicing technique).

1.4. Contribution

The major contribution of the research is a model slicing platform that provides implementations of two model slicing techniques. 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 slicing of object configurations and class models including invariants;

(3) A rigorous technique that supports slicing of class models including both invariants and operation contracts;

(4) A research prototype that provides implementations of two proposed class model slicing techniques;

(5) A framework for evaluating the effectiveness and the correctness of the proposed slicing techniques.

1.5. 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 provides a tool support for the proposed slicing techniques. 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 2014.

Kitchenham et al. [23] described a systematic procedure to identifying and analyzing available literature relevant to a specific research topic. Tao [53] 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 [53]. 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 [15] and can have invariants and operation contracts expressed using the OCL [43]. 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 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

Table 2.1. Electronic Search Result (from 2003 to 2014)

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

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 2014 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 2014 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 Engineering (ASE), IEEE International Conference on Software Maintenance (ICSM), IEEE International Requirements Engineering Conference (RE), and International Conference on Fundamental Approaches to

Table 2.2. Manual Search Result from Journals (from 2003 to 2014)

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

Table 2.3. Manual Search Result from Conferences (from 2003 to 2014)

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

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 decision (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][7][8] [20][22][26][27][38][39][40][41]) 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][42]) 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, [39][40][41][42], 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 slicing

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 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. [38] 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. [7][8] 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 elements 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. [39][40][41][42] 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

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	[38]	Metamodel pruning	Basic	Classes and properties	No	Automated
4	[7][8]	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	[39][40] [41][42]	Analysis	Inv	Invariants	Dependency graph	Automated

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.

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 chapter we provide background material needed to understand the proposed slicing techniques described in the disertation. In this dissertation we focus on two types of class model analysis: one (*Invariant Checking*) that involves checking conformance between an object configuration and a class model with specified invariants, and the other (*Contract Checking*) that analyzes sequences of operation invocations to uncover invariant violations. Section 3.1 describes a list of tools that can be used for invariant checking. Section 3.2 describes an approach we developed to tackle the contract checking.

3.1. Invariant Checking Tools

USE [17][36], developed by the Database Systems Group at Bremen University, is a modeling tool for specifying object-oriented systems. Figure 3.1 shows an overview of the USE tool. It allows developers to generate object models 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 [15] and the Object Constraint Language (OCL) [43]. 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 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

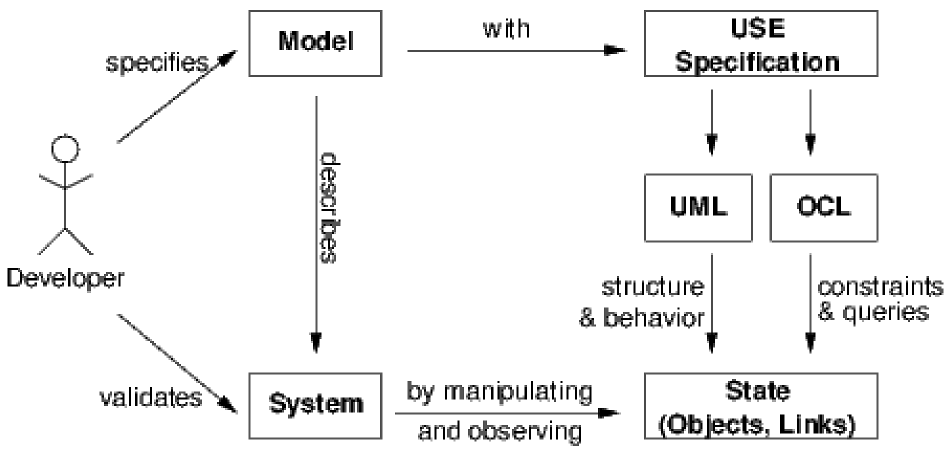


FIGURE 3.1. USE Tool Overview (excerpted from the USE project [17])

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 for invariant checking [29]. 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 models can be specified using Alloy predicates. If the Alloy Analyzer cannot return an instance of an Alloy model for a predicate specifying an object model within a bounded scope, the object model specified using the predicate may not be consistent with a class model expressed using the Alloy model.

The Kermeta language [14][21][31] was developed by Triskell Team at INRIA. It is an executable metamodeling language implemented on top of the Eclipse Modeling Framework (EMF) [44] 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 checker [46] is an implementation of the OCL OMG standard [43] 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 [44]. The extensibility of the provided APIs allows software modelers to develop their own customized prototypes for a variety of OCL analysis tasks. The evaluation framework described in the dissertation also builds upon the Eclipse OCL checker.

3.2. Analyzing Operation Contracts of UML Class Models

In previous work [45] 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

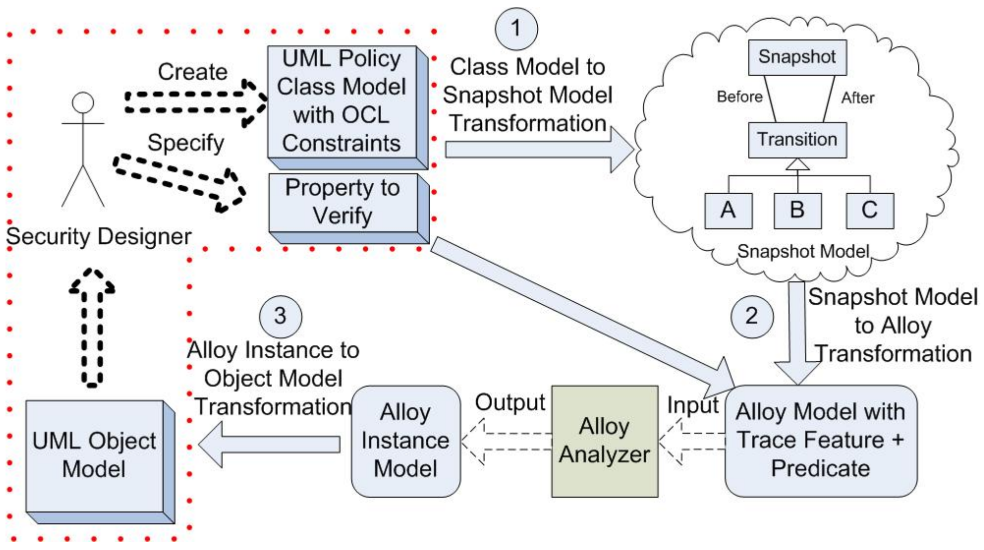

1
Class Model to Snapshot Model Transformation
Create
Specify
UML Policy Class Model with OCL Constraints
Property to Verify
Security Designer
Snapshot
Before
After
Transition
A
B
C
Snapshot Model
2
Snapshot Model to Alloy Transformation
3
Alloy Instance to Object Model Transformation
UML Object Model
Alloy Instance Model
Output
Alloy Analyzer
Input
Alloy Model with Trace Feature + Predicate

FIGURE 3.2. Class Model Analysis Approach Overview

in the approach build upon the UML2Alloy transformation tool [9][2][1] developed at the University of Birmingham. The work proposed in [45] 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 [45] also builds upon our previous work on the Scenario-based UML Design Analysis (ScUDA) approach [52][50][51][49]. 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 [45] 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 [45] is used to answer the question "Is there a scenario 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 [52] 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 describe the slicing techniques that can be used for invariant checking and contract checking. The first slicing technique can be used to improve the scalability of the invariant checking that involves checking conformance between an object configuration and a class model with specified invariants, and the second slicing technique can be used to improve the scalability of the contract checking that involves analyzing sequences of operation invocations to uncover invariant violations.

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

4.1. Co-slicing Class Model and Object Configuration

In this section we present the motivation for using the slicing technique to improve the efficiency of the invariant checking approach (Section 4.1.1), use a motivating example to illustrate the role of the slicing technique in the context of invariant checking (Section 4.1.2), describe the slicing technique (Section 4.1.3), and provide a discussion of the slicing technique (Section 4.1.4).

4.1.1. Motivation. At design time, a typical invariant checking process is characterized below: (1) A software modeler creates a class model; (2) He uses a tool (e.g., reverse engineering tools) to create a set of object models that conform to the class model; (3) He specifies a list of Well Formed Rules (WFRs) (i.e., invariants) and checks models against the WFRs using invariant checking tools such as the Eclipse OCL checker [46]. The entire process works well for small object models with hundreds of elements, and the modeler can receive the feedback from the checking tools within seconds or minutes.

However, given the growing complexity of software systems, models used to represent these complex systems will also grow significantly in size. While design models were built by hand in the early days of MDD, nowadays models with possibly more than one million of elements can be built programmatically. For example, we used a reverse engineering tool, namely MoDisco [10], to generate Java models from multiple Eclipse platform plugins. The object models generated from Eclipse plugins could have up to one million elements. The checking time significantly goes up (e.g., more than two hours in the worse case scenario) for large models with hundreds of thousands of elements. This motivates the use of the scalable invariant checking approach in the context of large object models.

Checking object models against invariants does not need the entire class model and the full object model to be present. Instead only a small part of the class and object models that are referenced by the invariants needs to be used for the invariant checking. In addition, a substantial number of invariants only reference part of the class model in which they are defined [11]. This motivates the use of the slicing technique in the context of invariant checking. The slicing technique thus can be used to reduce the size of the input class and object models to make the checking more efficient.

4.1.2. Motivating Example. Figure 4.1 shows part of a class model that describes the information system of a bus company (excerpted from [39]). In the class model, a trip uses more than one coach. A coach is controlled by multiple security guards. A passenger can buy multiple tickets from a vending machine located 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 manager. Figure 4.2 shows a valid instance of the class model given in Figure 4.1.

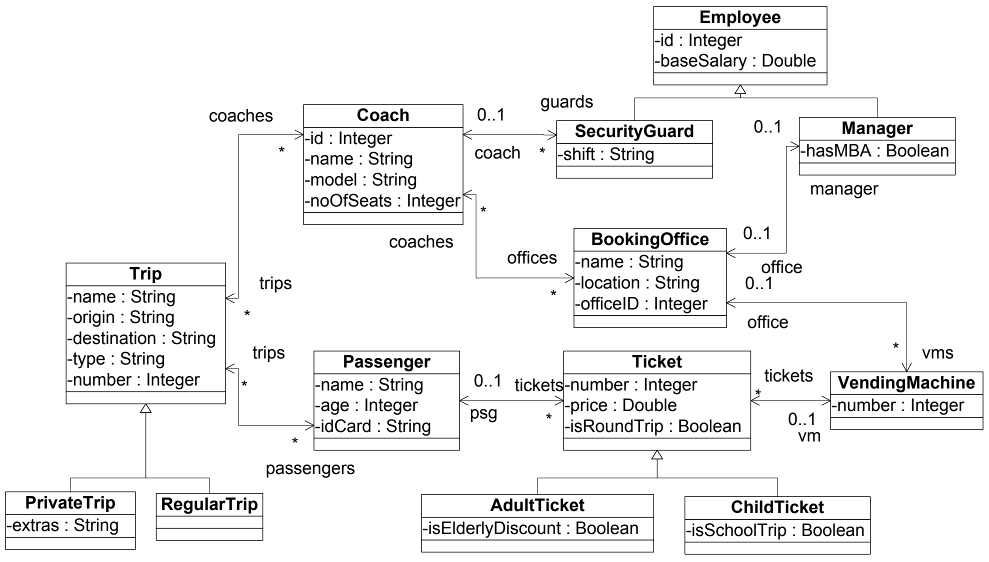


FIGURE 4.1. A class model (expressed using the class diagram notation) that describes the information system of a bus company

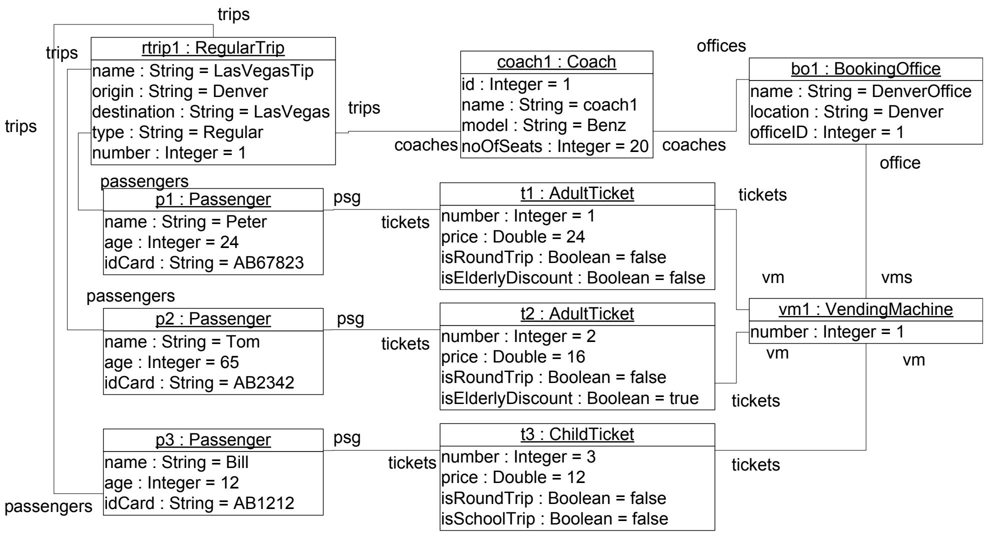


FIGURE 4.2. An object model (expressed using the object diagram notation) conforming to the class model given in Figure 4.1

Invariants defined in the class model are given in Table 4.1. For example, the *UniqueTicketNumber* invariant is defined in the context of the *Ticket* class in the ckass model, and it specifies that every ticket must have a unique number. Such invariants cannot be expressed directly using the class diagram notation, and thus are specified using other languages such

Table 4.1. A list of invariants in the class 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$\rightarrow$forAll(t | t.passengers$\rightarrow$size()$\leq$ noOfSeats)

// Each ticket must have a unique number. **Context** *Ticket* **inv UniqueTicketNumber:** Ticket::allInstances()$\rightarrow$forAll(t1, t2 | t1.number = t2.number implies t1 = t2)
// Each regular trip must have more than six passengers. **Context** *RegularTrip* **inv MinPassengers:** self.passengers$\rightarrow$size() $\geq$ 6

as the OCL [43]. The model in Figure 4.2 may or may not satisfy the invariants defined in the class model.

Tools, such as the Eclipse OCL checker [46], can be used in this situation for invariant checking. They take as input the invariant, the class model and the object model, and produce checking results, indicating whether the object model is consistent with the invariant defined in the class model. In this case, the object model in Figure 4.2 does not violate the *UniqueTicketNumber* invariant defined in the class model since *t1, t2, t3* have different numbers (i.e., 1, 2, 3).

Note that to check the *UniqueTicketNumber* invariant, we only need to look into the tickets and their numbers. The rest of object model is not relevant to the checking. Indeed, the *UniqueTicketNumber* invariant is defined in the context of the *Ticket* class and only refers to the *number* attribute in the *Ticket* class. Therefore, instead of feeding the entire class model in Figure 4.1 and the full object model in Figure 4.2 into the invariant checking tools, we can use the slicing technique to generate smaller class and object models. Figure 4.3 shows the sliced class and object models that are parts of the original class and object models. We can feed these generated class and object models into the tools for invariant

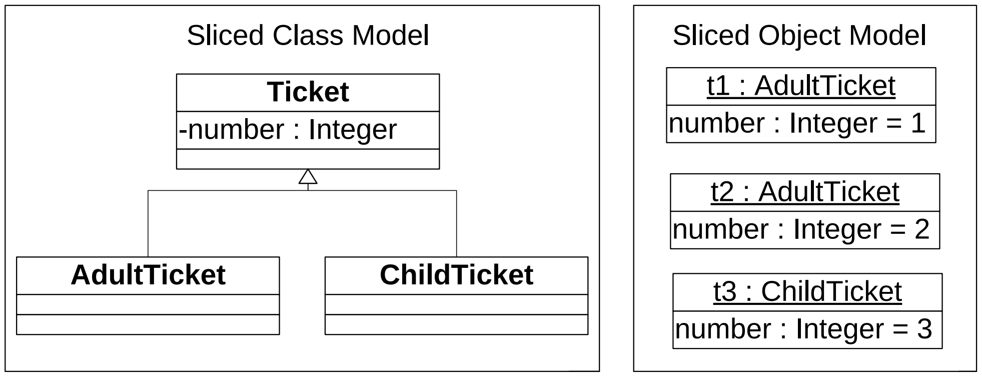


FIGURE 4.3. Checking the *UniqueTicketNumber* invariant in the context of sliced class and object models

checking. Note that we keep the classes *AdultTicket* and *ChildTicket* in the generated class model since *UniqueTicketNumber* is defined in the context of the *Ticket* class and thus can be inherited by the subclasses of the *Ticket* class. Therefore the instances of *AdultTicket* and *ChildTicket* also need to be checked against the *UniqueTicketNumber* invariant.

It is also important to note that the slicing technique is needed for invariant checking only if (1) the use of the slicing technique can reduce the invariant checking time (i.e., effectiveness of the slicing technique), and (2) the checking results remain the same in the context of the slicing technique (i.e., the correctness of the slicing technique). For example, the slicing technique is needed for checking the *UniqueTicketNumber* invariant if (1) the checking time for the class and object models in Figure 4.3 is less than that used for original class and object models, and (2) the invariant checking results for the original models are the same as the results for the models in Figure 4.3. We have conducted an evaluation to explore the effectiveness and the correctness of the slicing technique in the context of invariant checking, and the evaluation results are given in the Evaluation Chapter.

4.1.3. SLICING TECHNIQUE FOR INVARIANT CHECKING. Figure 4.4 shows an overview of the invariant checking approach in the context of the slicing technique. The input of the checking includes a class model (*MM*), an object model (*M*), and one or many OCL invariants

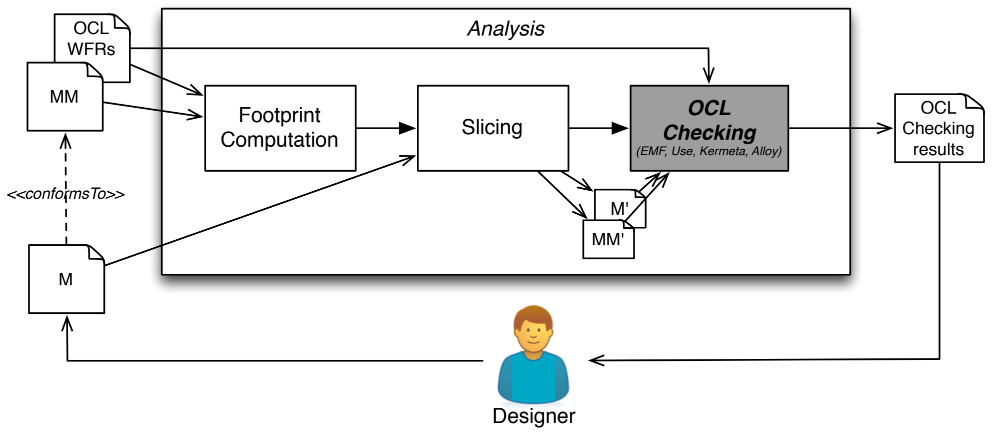


FIGURE 4.4. An overview of the invariant checking approach in the context of the slicing technique

(*Well-Formed Rules*). First, the approach computes a footprint from the class model and OCL invariants. The concept of the footprint is from [20], where a footprint refers to part of a class model that contains all elements that affect the outcome of an operation. In this paper a footprint refers to all class model elements that are directly referenced by the input OCL invariants. Second, the footprint serves as slicing criterion, and is used to generate a sliced class model (*MM'*) from the input class model. The sliced class model (*MM'*) includes (1) all the class model elements from the footprint, and (2) all the subclasses of the classes in the footprint. Third, the sliced class model (*MM'*) is used to generate a sliced object model (*M'*) from the input object model. The sliced object model (*M'*) contains only object model elements that are instances of class model elements in *MM'*. Finally, the sliced class and object models with the invariants are fed into the tools for invariant checking.

In the remainder of this section we illustrate the slicing technique step by step.

4.1.3.1. *Generating Footprint.* Table 4.2 shows the footprint of each invariant given in Table 4.1. The footprint of an invariant contains class model elements such as classes, attributes, references, and/or enumerations. The footprint computation takes as a class model and an invariant, and analyzes the dependencies between the invariant and the class

TABLE 4.2. Footprint (i.e., dependent elements) of each invariant (see Table 4.1) defined in the class model in Figure 4.1

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

model. The dependency analysis is performed by traversing the syntax tree of the OCL invariant.

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

Note that the *MinPassengers* invariant depends on both *Trip* and *Passenger* classes. This is because *MinPassengers* uses the *passengers* reference in its definition (see Table 4.1). Reference *passengers* is defined in the context of class *Trip* and can be inherited by the subclasses (e.g., *RegularTrip*) of *Trip*. In addition, the type of the *passengers* reference is class *Passenger*.

Algorithm 1 Slice a class model

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

4.1.3.2. *Slicing Class Model.* Algorithm 1 is used to generate a sliced class model (*MM'*) from a footprint (*FP*) and the original class model (*MM*). The slicing criteria in this case would be the class model elements in the footprint. Algorithm 1 computes *Subs*, a set of classes that are the subclasses of the classes in the footprint. The sliced class model contains only elements that are from the footprint and *Subs*. The reason we keep these referred classes' subclasses in the sliced class model is that the instances of subclasses are also the instances of their super classes, and thus can be used for invariant checking.

Consider the case in which a modeler wants to check whether the model in Figure 4.2 satisfies the *MaxCoachSize* invariant defined in the class model in Figure 4.1. Algorithm 1 can be used in this case to slice the class model in Figure 4.1 using the footprint of the *MaxCoachSize* invariant. Figure 4.5 shows the sliced class model that is generated from the footprint of the *MaxCoachSize* invariant. Since the footprint of the *MaxCoachSize* invariant includes class *Trip*, and class *Trip* has two subclasses, *PrivateTrip* and *RegularTrip*, the *MaxCoachSize* invariant also depends on *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 $noOfSeats$.

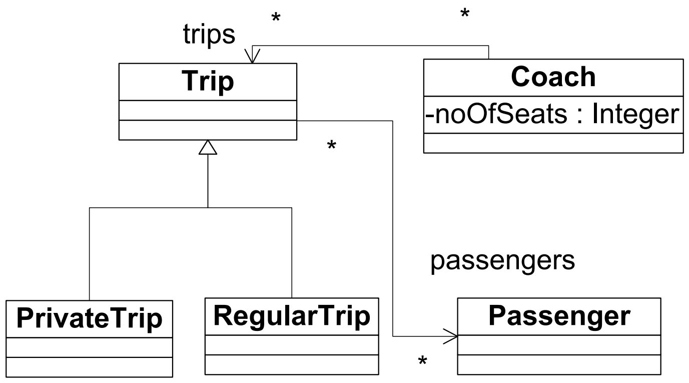


FIGURE 4.5. A class model slice generated from the footprint of the *MaxCoachSize* invariant

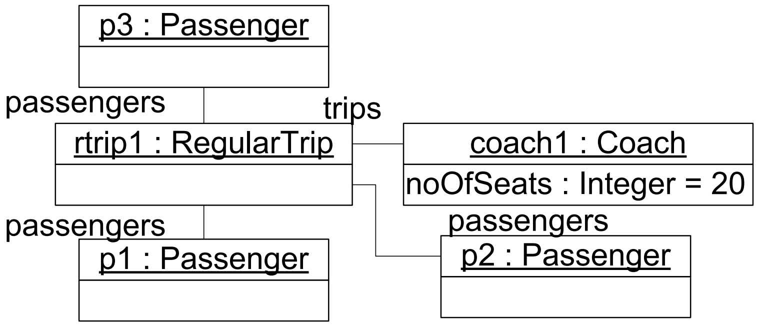


FIGURE 4.6. An example of a sliced object model

4.1.3.3. *Slicing Object Model.* Algorithm 2 is used to slice an object model. It takes as input an object model (*M*) and a sliced model (*MM'*), and produces a sliced object model, where each element in the sliced object model is an instance of a class model element in the sliced class model. For example, given the sliced class model in Figure 4.5 and the object model in Figure 4.2, Algorithm 2 can be used to generate a sliced object model (see Figure 4.6) that conforms to the sliced class model in Figure 4.5. Note that the sliced object model is a valid instance of the sliced class model, but it may or may not satisfy the invariants used to generate the sliced class model.

The algorithm checks each object (see lines 5-7) in *M*, and removes an object and its slots/link ends if the object's metaclass is not in the set of classes involved in *MM'*. For example, object *bo1:BookingOffice* in Figure 4.2 is not an instance of any class involved in the sliced class model in Figure 4.5, and thus can be removed from the object model. In addition, its slots (i.e., *name*, *location*, and *officeID*) and link ends (i.e. *coaches* and *vms*) are also removed from the object model. Note that both slots and link ends are the modeling concepts used in the object diagram notation, where slots are the instances of attributes and link ends are the instances of references.

Algorithm 2 also checks the slots and link ends of each unremoved object. Lines 9-13 remove a slot of an object if the slot's corresponding attribute is not included in *Attrs*, a set of attributes in *MM'*. Lines 14-18 remove a link end of an object if the link end's corresponding reference is not included in *Refs*. For example, object *rtrip1:RegularTrip* in Figure 4.2 has 5 slots and 4 link ends. Since class *RegularTrip* and its super class *Trip* in Figure 4.5 have no attributes, the slicing algorithm removed all the slots from object *rtrip1:RegularTrip* as indicated in Figure 4.6. Similarly, class *RegularTrip* and its super class *Trip* in Figure 4.5 have only one reference, *passengers*. Thus, the slicing algorithm removed link end *coaches* from object *rtrip1:RegularTrip*.

4.1.4. Discussion. The slicing technique described in Section 4.1 can have different variations based on (1) the number of input invariants (Section 4.1.4.1), (2) the way of footprinting (Section 4.1.4.2), (3) the way of slicing the object model (Section 4.1.4.3), and (4) the checking context (Section 4.1.4.4).

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

4.1.4.1. *Single v.s. Multiple Invariants.* The slicing technique described in Section 4.1 can be used for a single invariant or multiple invariants. The illustration example given in Section 4.1 shows how the slicing technique handles single invariant input. If the input includes multiple invariants (e.g., *N* invariants), the slicing technique generates a sliced class model and a sliced object model for each invariant, and performs the checking *N* times. Thus

Algorithm 2 Slice an object model

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

the problem of checking multiple invariants can be reduced to the problem of checking single invariant.

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

It is also possible to (1) produce a single invariant from multiple invariants using conjunction, and (2) checking the conjunctive invariant against the object model. However, invariants conjunction may cause two problems. First, the accuracy of the checking results would be decreased. For example, if an object model violates the conjuncted invariant, it is not clear about which single invariant is violated by the object model. Second, checking the conjuncted invariant would be computationally different from (or worse than) checking multiple invariants one by one. For example, one would use the operation *allInstance()* to merge two invariants with different contexts (e.g., see the *MinCoachSize* and *MinPassengers* invariants in Table 4.1) into one conjuncted invariant (see the *ConjunctedInv* invariant below).

Context *RegularTrip* **inv ConjunctedInv:**

self.passengers→size() ≥ 6 and Coach.allInstances()→forAll(c|c.noOfSeats ≥ 10)

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

4.1.4.2. *Static v.s. Dynamic Footprinting.* The way of footprinting can be categorized into two types: static footprinting and dynamic footprinting [20]. Static footprinting uses only the information from the invariant to guide the footprinting process. The model footprinting described in Section 4.1 is an example of static footprinting [20]. Dynamic footprinting uses the elements in object models to identify an invariant that is applicable to the object models (referred to as checking relevant invariant). The intuition behind this is based

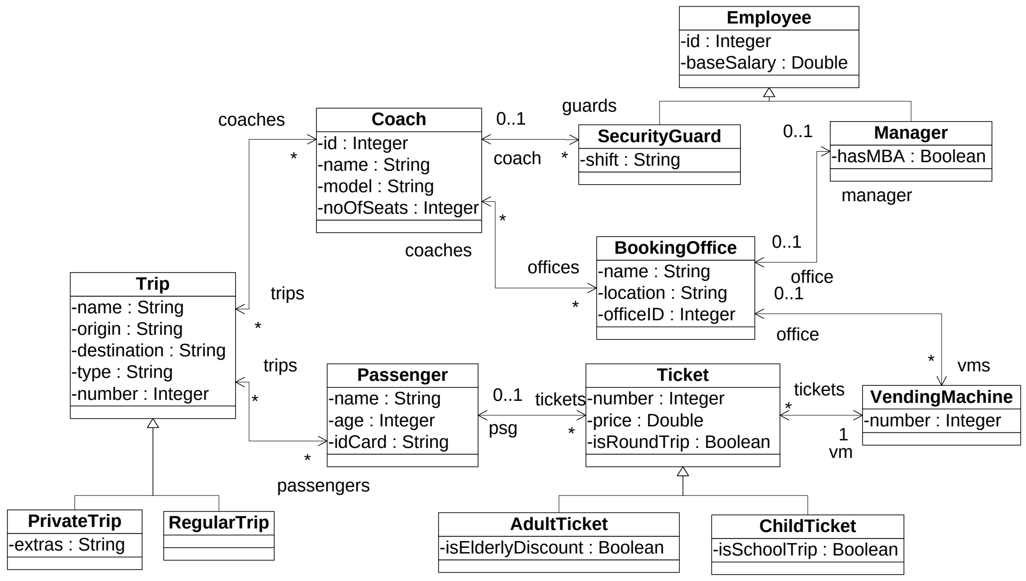


Figure 4.7. A modified version of the class model in Figure 4.1 (the multiplicity of the *vm* reference has been changed)

on the following observation: If each element in an object model is not an instance of any class model element that is referred by an invariant, there is no need to check the object model against the invariant because the object model will not violate the invariant.

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

4.1.4.3. *Aggressive v.s. Conservative Class Model Slicing.* In aggressive slicing, the sliced class model only contains the elements in the footprint (i.e., slicing criterion) and the subclasses of the classes in the footprint. In conservative slicing (also called pruning [38]), the sliced class model also contains the classes that are types of the mandatory references (i.e., the lower bound of the multiplicity of the reference must be equal to or larger than 1) from the classes in the footprint.

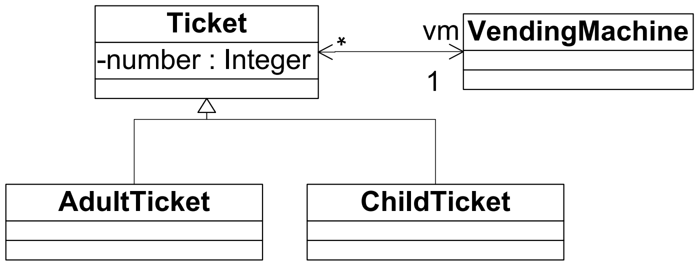


FIGURE 4.8. A sliced class model generated from the *UniqueTicketNumber* invariant and the class model in Figure 4.7 using the conservative slicing

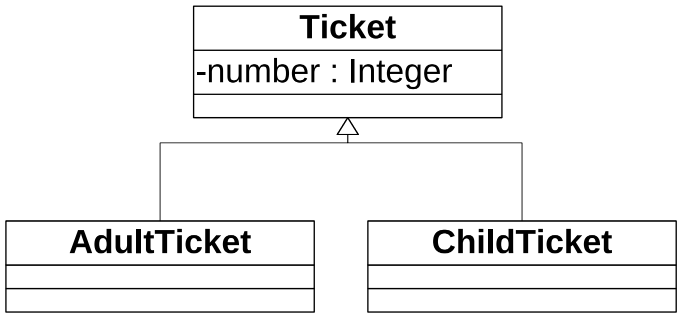


FIGURE 4.9. A sliced class model generated from the *UniqueTicketNumber* invariant and the class model in Figure 4.7 using the aggressive slicing

For example, Figure 4.7 shows a modified version of the class model in Figure 4.1. Note that the multiplicity of the *vm* reference in Figure 4.1 is 0..1, while the multiplicity of the *vm* reference in Figure 4.7 is 1. The *vm* reference in Figure 4.7 is an example of the mandatory reference.

Given the *UniqueTicketNumber* invariant and the class model in Figure 4.7, the class model (see Figure 4.9) generated using the aggressive slicing contains only the *Ticket* class and its subclasses, while the class model (see Figure 4.8) generated using the conservative slicing contains the mandatory reference (i.e., *vm*) and its type class (i.e., *VendingMachine*).

The slicing technique described in Section 4.1 uses the aggressive slicing. This is mainly because the aggressive slicing could produce smaller class and object models and thus reduce the checking time. However, the aggressive slicing could be a threat to the correctness of the slicing technique if the input object model is not a valid instance of the input class model.

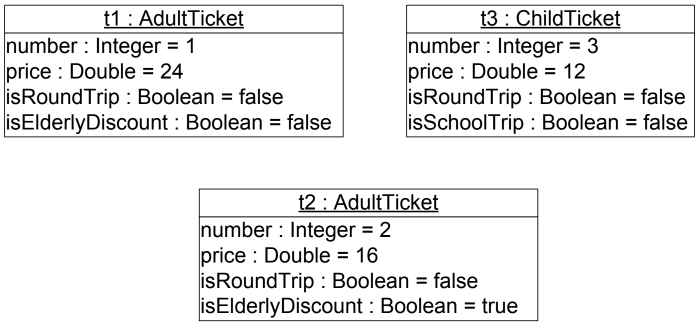

t1 : AdultTicket
number : Integer = 1
price : Double = 24
isRoundTrip : Boolean = false
isElderlyDiscount : Boolean = false
t3 : ChildTicket
number : Integer = 3
price : Double = 12
isRoundTrip : Boolean = false
isSchoolTrip : Boolean = false
t2 : AdultTicket
number : Integer = 2
price : Double = 16
isRoundTrip : Boolean = false
isElderlyDiscount : Boolean = true

FIGURE 4.10. An object model that is not a valid instance of the class model in Figure 4.7

For example, Figure 4.10 shows an object model that is not a valid instance of the class model in Figure 4.7. Given the object model in Figure 4.10 and the class model in Figure 4.7, the checking results should return false since a ticket needs to be associated with a vending machine (see the multiplicity of the *vm* reference). However, if we use the class model in Figure 4.9 (i.e., the result of the aggressive slicing for the *UniqueTicketNumber* invariant) to slice the object model in Figure 4.10, and check the sliced object model against the *UniqueTicketNumber* invariant in the context of the class model in Figure 4.9, the checking results would return true, which contradicts the checking results without using the slicing technique.

If we use the class model in Figure 4.8 to slice the object model in Figure 4.10, and check the sliced object model, the checking results would return false (see the multiplicity of the *vm* reference), which is consistent with the checking results without using the slicing technique. Therefore, to use the aggressive slicing, we need to add a precondition to the slicing technique, that is, the input object model must be a valid instance of the input class model.

4.1.4.4. *Checking in the Context of the Entire Class Model v.s. the Sliced Class Model.* In the slicing technique described in Figure 4.1, the sliced object model is checked in the context of the sliced class model. This is mainly because of two reasons. First, the sliced class model is smaller than the entire class model, and the invariant checking in the context of the sliced class model would take less time. Second, the aggressive slicing could produce object models that are not valid instances of the entire class model, and the checking results for such object models in the context of the entire class model always return false.

For example, given the *UniqueTicketNumber* invariant, the object model in Figure 4.2 and the class model in Figure 4.7, the aggressive slicing technique would produce a sliced class model in Figure 4.9 and a sliced object model in Figure 4.10. When the sliced object model is checked in the context of sliced class model, the checking results would be true (every ticket has a unique number) which is consistent with the checking result without using the slicing technique. However, if the sliced object model is checked in the context of the entire class model in Figure 4.7, the checking result would be false, because a ticket needs to be associated with a vending machine, and the sliced object model is not a valid instance of the class model in Figure 4.7. Therefore, to check the sliced object model in the context of the entire class model, we need to use the conservative slicing for the slicing technique.

4.2. Slicing a Class Model with OCL Constraints

The model slicing technique described in this section is used for contract checking. It decomposes a large class model into fragments, where each fragment contains model elements needed to analyze a subset of the invariants and operation contracts in the class model. Figure 4.11 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)

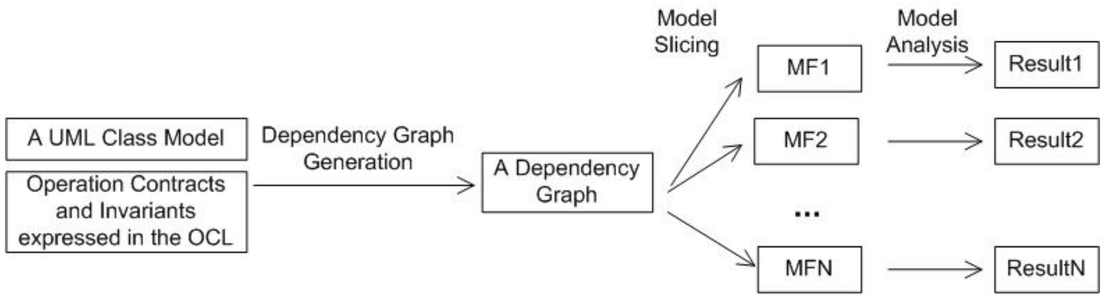


FIGURE 4.11. Technique Overview

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, the slicing algorithm used to decompose the class model into model fragments, and a discussion of the slicing technique.

4.2.1. ILLUSTRATING EXAMPLE. We will use the Location-aware Role-Based Access Control (LRBAC) model, proposed by Ray et al. [33] [34] [35], to illustrate the model slicing technique. LRBAC is an extension of Role-Based Access Control (RBAC) [37] that takes location into consideration 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

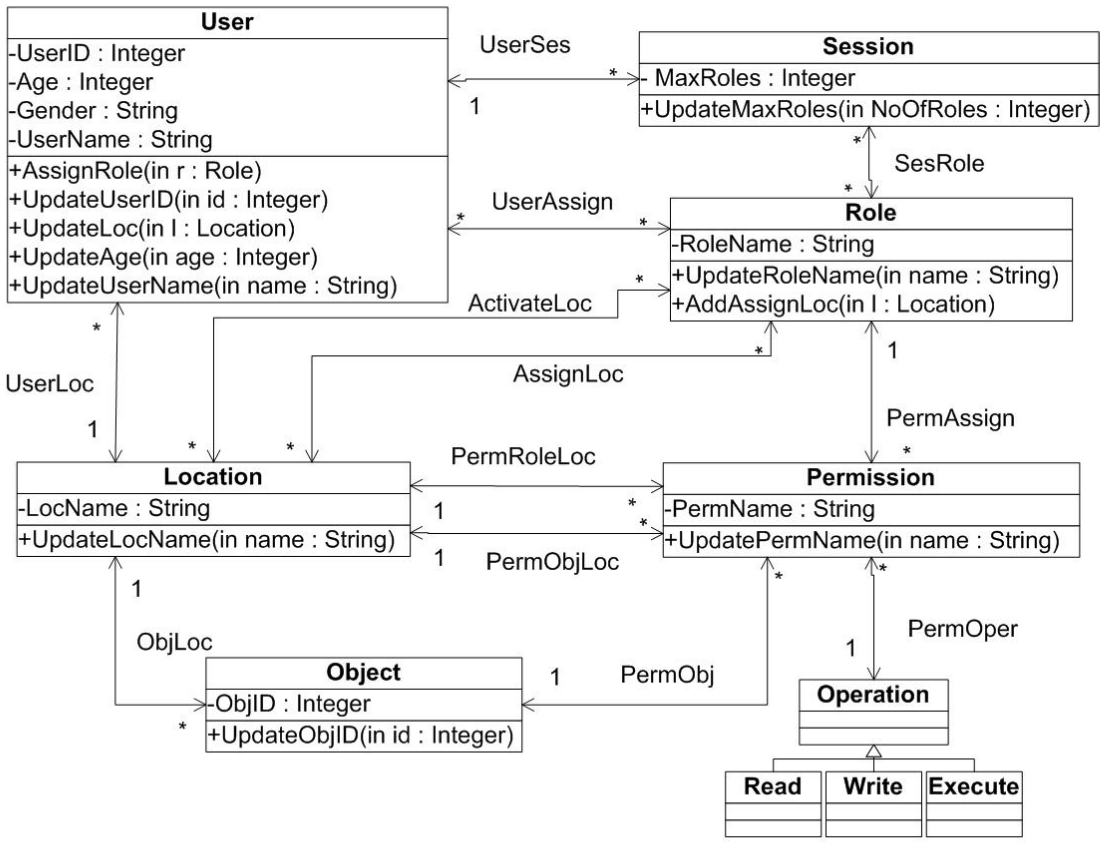


FIGURE 4.12. A Partial LRBAC Class Model

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 4.12 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 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.

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

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 [45], we developed a class model analysis 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

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

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.

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 $self.UserAssign$

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)

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 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

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

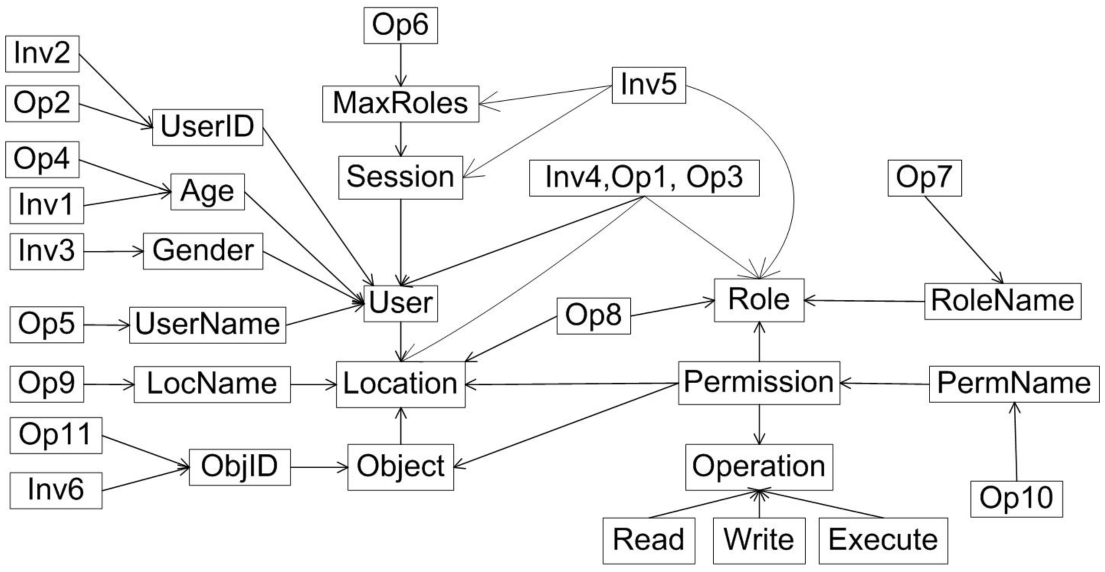


Figure 4.13. A Dependency Graph

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 4.13 shows a graph that describes the dependency relationship among classes, attributes, operations and invariants of the LRBAC class model described in Fig. 4.12. Algorithm 3 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. 4.13. 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. 4.13 since invariant $Inv5$ also references class *Role* in its specification through the navigation from class *Session* to class *Role*.

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

Algorithm 3 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 ($cls$) and attribute ($attr$) of an operation contract ($op$) or invariant ($inv$), create a directed edge to vertex $cls$ and $attr$ from vertex $op$ or $inv$. If the operation contract or invariant only references its context class 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.
```

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 4, 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 [39]. Model elements that are involved in a *local* analysis problem are referred to as *local analysis model elements*. For example, operation *UpdateUserID* in Fig. 4.12 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

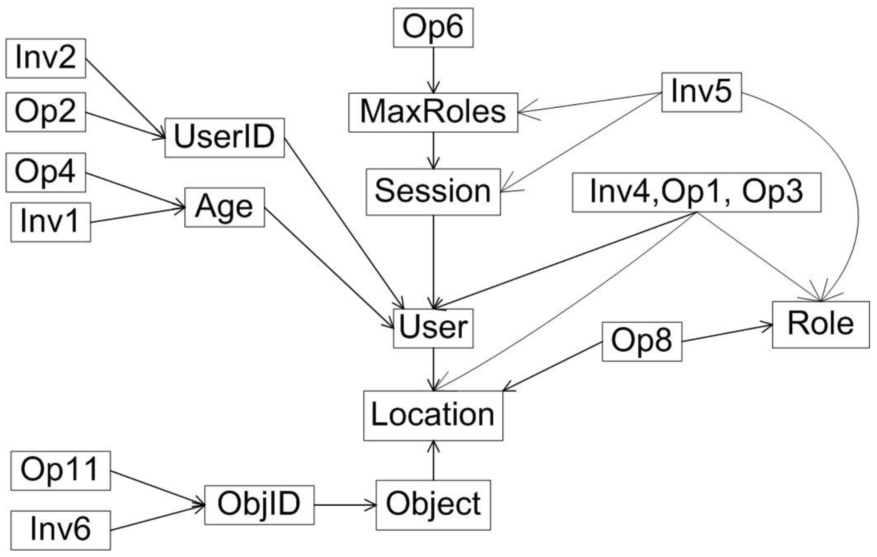


FIGURE 4.14. A Dependency Graph Representing a LRBAC Model with the *Irrelevant Model Elements* Removed

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 $Inv2$ 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 5. 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 6.

4.2.3.1. *Identifying* Irrelevant Model Elements*:* Algorithm 4 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 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 4.14 shows a dependency graph representing a LRBAC model with the analysis irrelevant model elements removed. Lines 6-14 of Algorithm 4 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 $ARVSet$.

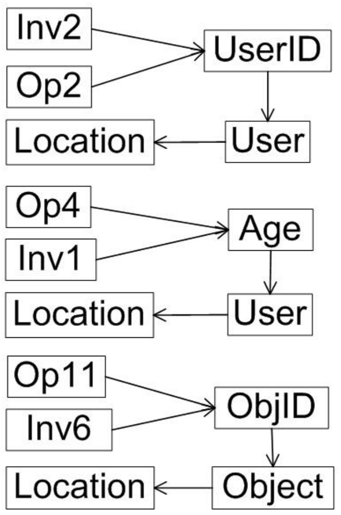
4.2.3.2. *Identifying* Local Analysis Model Elements*:* Algorithm 5 is used to identify fragments representing local analysis problems. The algorithm first computes a set of attribute and class vertices, $LocalVSet$, that are involved in the local analysis problems. A vertex, $ClsAttrV$, is added to $LocalVSet$ 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 problem.

Algorithm 4 Irrelevant Model Elements Identification Algorithm

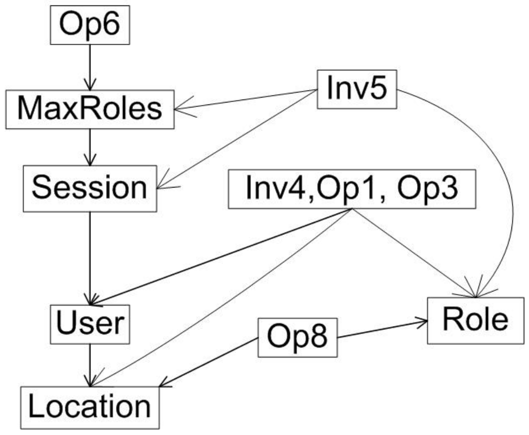
```
1: Input: A dependency graph
2: Output: A set of analysis-irrelevant vertices
3: Algorithm Steps:
4: Set $OpVSet$ = a set of operation vertices, $InvVSet$ = a set of invariant vertices;
5: Set $VSet$ = all the vertices in the dependency graph, $OpDVSet = \{\}$, $InvDVSet = \{\}$;
6: for each operation vertex $OpV$ in $OpVSet$ do
7:     Get a set of $OpV$'s directly dependent vertices, $OpVDDSet$;
8:     $OpDVSet = OpDVSet \cup OpVDDSet$;
9: end for
10: for each invariant vertex $InvV$ in $InvVSet$ do
11:     Get a set of $InvV$'s directly dependent vertices, $InvVDDSet$;
12:     $InvDVSet = InvDVSet \cup InvVDDSet$;
13: end for
14: Set $ARClsAttrVSet = OpDVSet \cap InvDVSet$, Set $AROpInvVSet = \{\}$;
15: for each vertex $V$ in $OpVSet \cup InvVSet$ do
16:     if one of $V$'s directly dependent vertex is in $ARClsAttrVSet$ then
17:         $AROpInvVSet = AROpInvVSet \cup V$; 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:     $ARVSet = ARVSet \cup VDFSSet$;
25: end for
26: Return ($VSet$ - $ARVSet$);
```

The dependency graph in Fig. 4.14 is decomposed into several subgraphs, as shown in Fig. 4.15a and Fig. 4.15b, using Algorithm 5. Each dependency graph in Fig. 4.15a 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. 4.14. 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



(A) Local Analysis Model Elements identified by Algorithm 5



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

FIGURE 4.15. Dependency Graphs Representing Model Fragments Extracted from the LRBAC Model in Fig. 4.12

to perform Algorithm 4 on DG to remove the analysis-irrelevant vertices, as indicated by Line 25.

4.2.3.3. *Decomposing the Dependency Graph:* Algorithm 6 is used to decompose a dependency graph without analysis-irrelevant vertices and local analysis problem related vertices. The algorithm computes a set of 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 6 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 4) 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. 4.15b is {*MaxRoles, User, Location, Role*}. Thus Col for the graph is $\{\{Op6, In5\}, \{In4, Op1, Op3\}, \{In4, Op1, Op3, Op8\}, \{In4, Op1, Op3, Op8, In5\}\}$.

Algorithm 5 Local Analysis Problem Identification Algorithm

```
1: Input: A dependency graph, $DG$, produced from the original graph after removing the
   irrelevant vertices produced by Algorithm 4
2: Output: A set of dependency graphs
3: Algorithm Steps:
4: Reuse $ARClsAttrVSet$ in Algorithm 4;
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:      $LocVSet = LocVSet \cup ClsAttrV$;
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 4 on $DG$ to remove analysis-irrelevant vertices;
```

Line 6 uses the union-find algorithm described in [12] 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. 4.15b 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 Figure 4.15b has only one disjoint set, the forest generated from the disjoint set consists of all the vertices in the dependency graph, indicating that

Algorithm 6 Dependency Graph Decomposition Algorithm

```
1: Input: A dependency graph, $DG$, produced from the original graph by Algorithm 5
2: Output: A set of dependency graphs
3: Algorithm Steps:
4: Recompute $ARClsAttrVSet$ for $DG$ using Algorithm 4;
5: Compute a collection $Col = S_1, S_2,...,S_v$ of operation and invariant vertex sets, where
   $S_v$ 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 [12] 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:      $SubVSet = SubVSet \cup VDFSSet$;
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 $DG$;
```

the graph in Figure 4.15b is the minimum dependency graph that cannot be decomposed further.

Figure 4.16 shows four model fragments extracted from the LRBAC model in Figure 4.12. Each model fragment corresponds to a dependency graph in Figure 4.15.

4.2.4. Discussion. The slicing technique described in the section 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 dissertation does not ensure that more than one independently analyzable fragments will be produced from a class model. However, an empirical study conducted by Juan et al. [11] showed that in practice a substantial number of invariants only reference

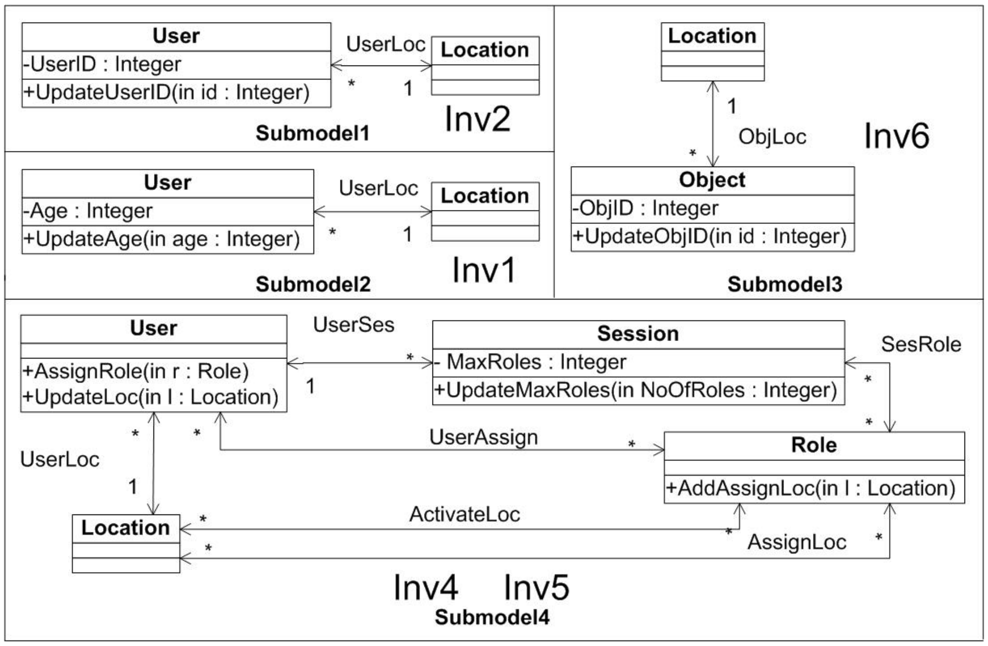


FIGURE 4.16. A List of Model Fragments Generated from the LRBAC Model in Figure 4.12

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.

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 [44] to parse Ecore class models and XMI object configurations.

(4) The prototype uses the APIs from Eclipse OCL project [46] 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).

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:

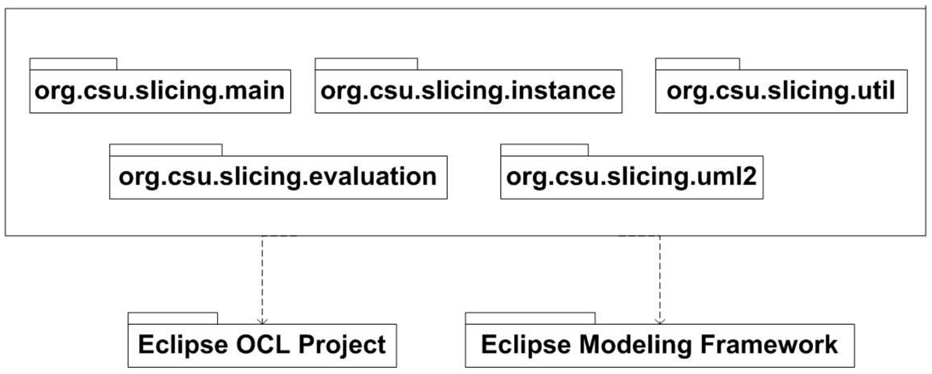


Figure 5.1. Prototype Architecture

(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.

(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:

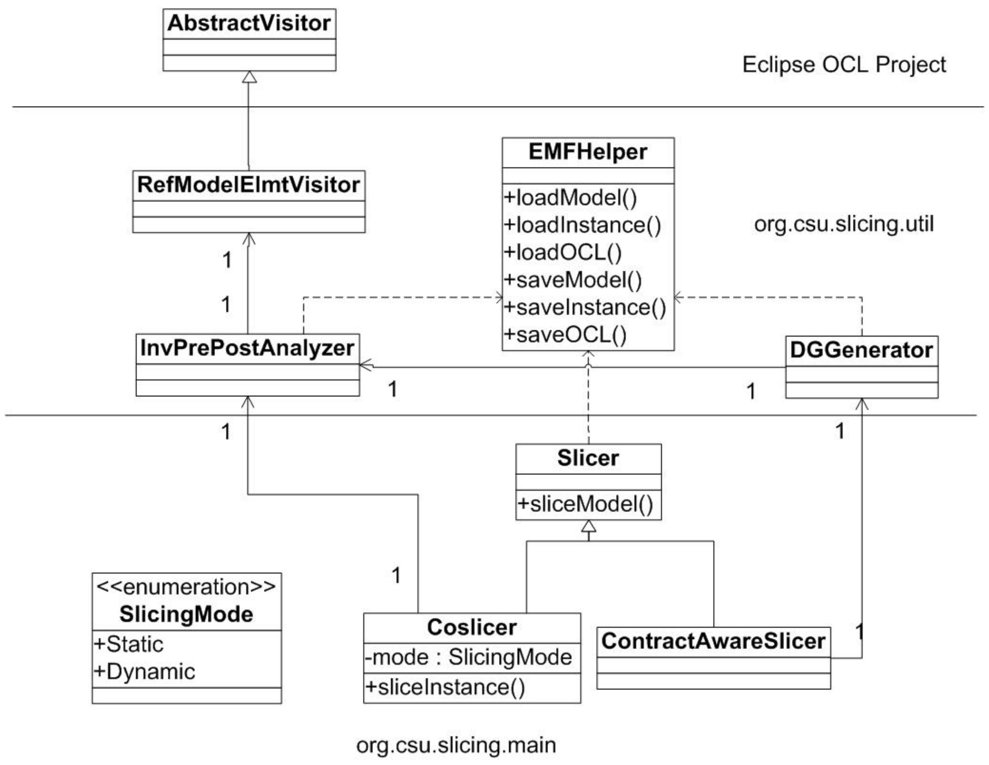


FIGURE 5.2. A partial UML class model that describes the implementations of the slicing techniques (part of classes, attributes, operation, and parameters are omitted)

(1) *AbstractVisitor*: The class is an abstract class from the Eclipse OCL project [46]. 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 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 used for the evaluation.

(2) *Ecore2XMI*: The class converts Ecore class models to XMI object configurations 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).

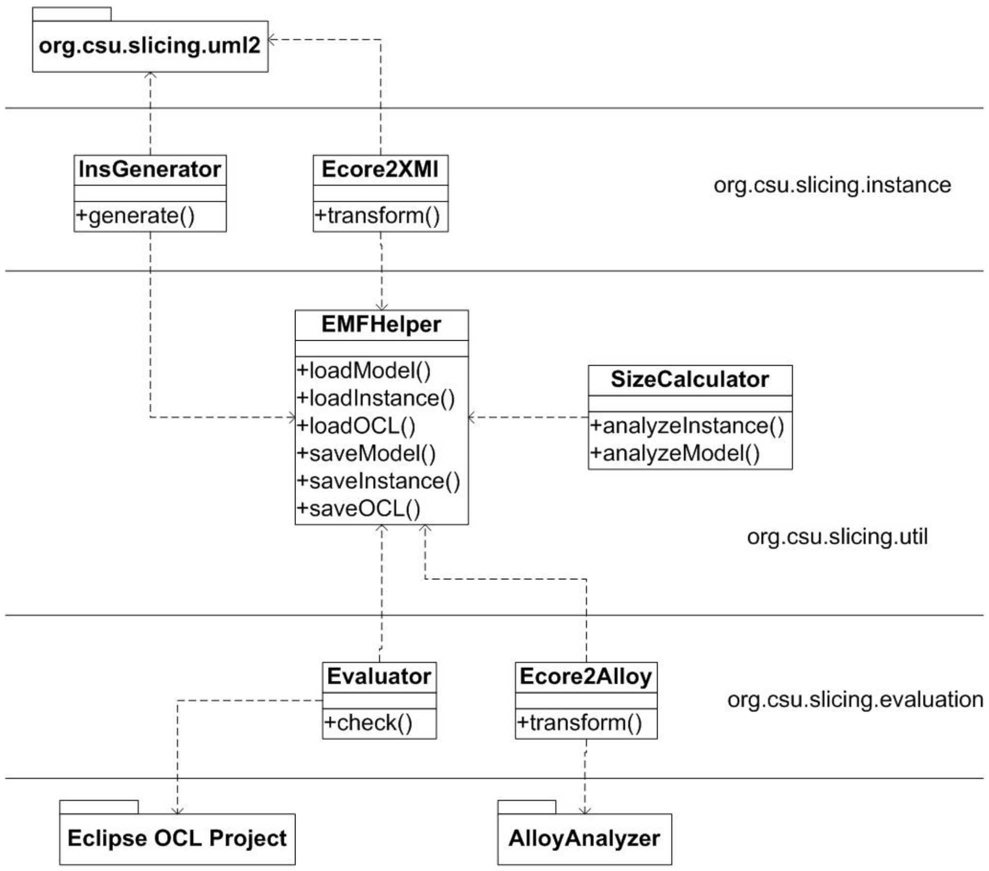


FIGURE 5.3. A partial UML class model that describes the implementation of the evaluation framework (part of classes, attributes, operation, and parameters are omitted)

(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 and correctness of the proposed slicing techniques. Specifically the evaluation aims to answer the following questions:

(1) *Q1*: Can the slicing technique improve the efficiency of the invariant checking and preserve the checking results?

(2) *Q2*: Can the slicing technique improve the efficiency of the contract checking and preserve the checking results?

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

Q1 can be divided into the following research questions:

RQ1.1 Can the slicing technique improve the efficiency of the invariant checking?

RQ1.2 Can the slicing technique preserve the invariant checking results?

To answer *RQ1.1*, we need to check whether the invariant checking time for unsliced class and object models (*CTUM*) is greater than the invariant checking time for sliced class and object models (*CTSM*). To avoid the bias of simply comparing the checking time of large and small models (i.e., unsliced and sliced models), we also need to take the time used to generate footprint and to slice class and object models (*ST*) into consideration.

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

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

If *CTS* is above 1, the slicing technique can improve the efficiency of the invariant checking. If *CTS* is below 1, the checking efficiency is not improved, and there is no need to use the slicing technique in the context of the invariant checking.

The slicing technique aims to improve the efficiency of the invariant checking. Thus it should not change the checking results. To answer *RQ1.2*, we need to check whether the invariant checking results for unsliced class and object models are the same as that for sliced class and object models.

The evaluation also provides the evaluation results for *RQ1.3*:

RQ1.3 Does the object model usage (i.e., the percentage of object model elements that are needed for the invariant checking) have negative impact on the checking time speedup (i.e., the lower the object model usage, the higher the checking time speedup)?

In *RQ1.3*, we are interested in exploring the relationship between the object model usage and the checking time speedup, that is, whether the decrease of the object model usage would make the checking time speedup increase. To answer *RQ1.3*, we need to calculate both the object model usage (see Metric 2) and the checking time speedup (see Metric 1). Metric 2 below is used to estimate the object model usage, η, where $\#_{sm}$ refers to the number of elements in the sliced object model and $\#_{um}$ refers to the number of elements in the unsliced object model:

$$\eta = \frac{\#_{sm}}{\#_{um}} \tag{2}$$

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

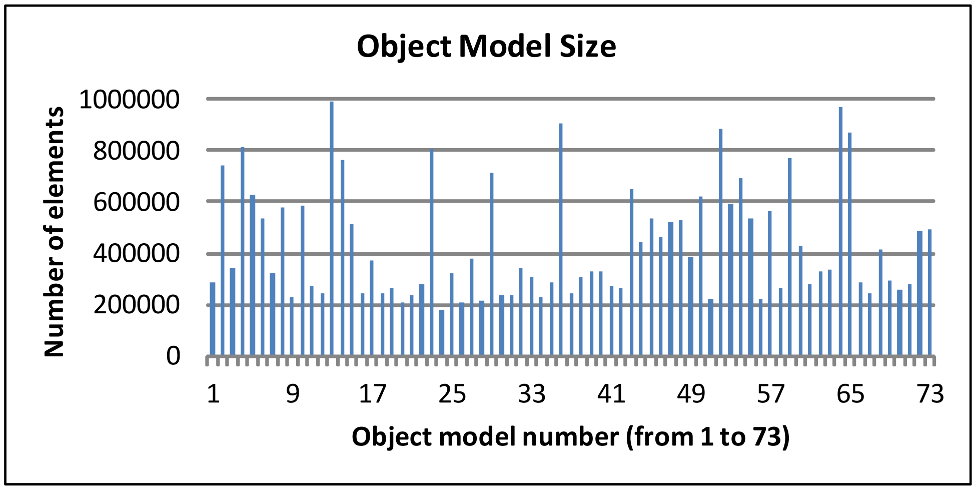


FIGURE 6.1. The sizes of the object models used in the evaluation

6.1.1. DATA COLLECTION. The class model used for the evaluation is the Java metamodel. The reason we chose Java metamodel is that (1) it is fairly complex (345 model elements including classes, attributes, references, and enumerations), and (2) its instances are relatively easy to collect from Java programs using reverse engineering tools. The object models used for the evaluation are generated from 73 Eclipse projects (Java source code). The reason we chose Eclipse projects is that (1) the Eclipse platform is widely used in both industry and academia, and (2) the object models generated from the Eclipse projects are quite large. We used a reverse engineering tool, namely MoDisco [10], to generate object models from the Java projects.

Figure 6.1 shows the sizes of the object models used in the evaluation. The sizes of these 73 models range from 175926 (i.e., object model generated from the *org.eclipse.gmf.runtime.draw2d.ui.render.awt* plugin) to 993319 (i.e., object model generated from the *org.eclipse.emf.ecore.xcore.ui* plugin) in terms of total number of object model elements including objects, links, and slots. Six invariants were used in the evaluation (see Table 6.1).

Table 6.1. A list of invariants used in the evaluation

Context *TagElement* **inv Inv1:** self.fragments→select(te | te.oclIsTypeOf(TextElement))→forAll(te1 | TextElement.allInstances()→exists(te2 | te2.text = te1.oclAsType(TextElement).text))
Context *SingleVariableAccess* **inv Inv2:** VariableDeclaration.allInstances()→exists(vd | vd = self.variable)
Context *Modifier* **inv Inv3:** (self.bodyDeclaration <> null implies BodyDeclaration.allInstances()→exists(bd | bd = self.bodyDeclaration)) and (self.singleVariableDeclaration <> null implies SingleVariableDeclaration.allInstances()→exists(svd | svd = self.singleVariableDeclaration)) and (self.variableDeclarationStatement <> null implies VariableDeclarationStatement.allInstances()→exists(vds | vds = self.variableDeclarationStatement)) and (self.variableDeclarationExpression <> null implies VariableDeclarationExpression.allInstances()→exists(vde | vde = self.variableDeclarationExpression))
Context *TypeAccess* **inv Inv4:** Type.allInstances()→exists(t—self.type.name = t.name)
Context *MethodInvocation* **inv Inv5:** MethodDeclaration.allInstances()→exists(md | md = self.method) and self.arguments→forAll(arg | arg.oclIsTypeOf(SingleVariableAccess) implies SingleVariableAccess.allInstances()→exists(sva | sva = arg))
Context *ExpressionStatement* **inv Inv6:** Expression.allInstances()→exists(e | e = self.expression)

6.1.2. Evaluation Results.

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

6.1.2.1. *Effectiveness of the Slicing Technique.* The evaluation framework takes as input the class model, 73 object models, and six invariants. For a pair of one object model and one invariant, the evaluation framework performs the following four steps. First, it checks the object model against the invariant in the context of class model, and measures the checking time (*CTUM*). Second, it generates a sliced class model for the invariant, slices the input object model using the sliced class model, and measures the time used to generate footprint and to slice the class and object models (*ST*). Third, it checks the sliced object model against the invariant in the context of the sliced class model, and measures the checking time (*CTSM*). Fourth, it calculates the *CTS* based on Metric 1. In total, there are 73 *CTS*s

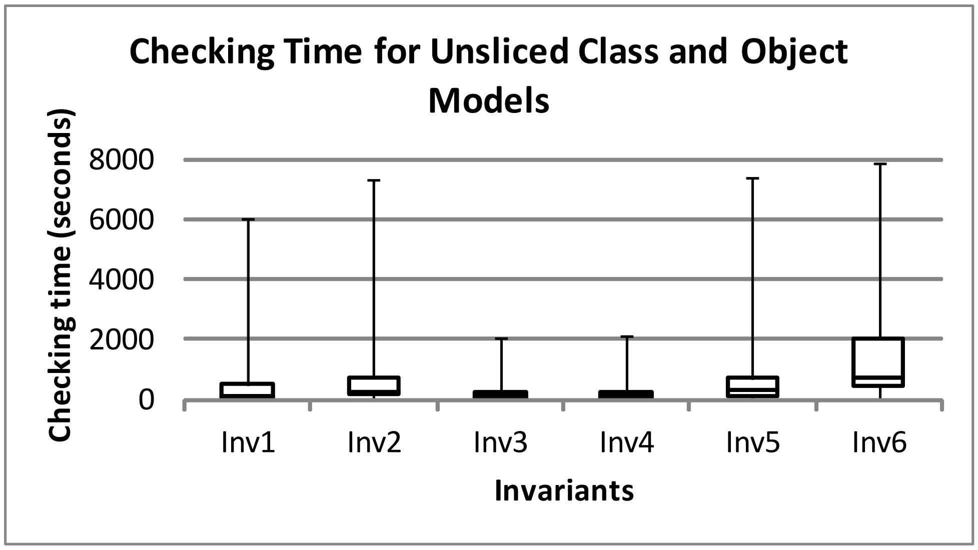


Figure 6.2. Box plot for the measurement of Checking Time for Unsliced class and object Models (*CTUM*)

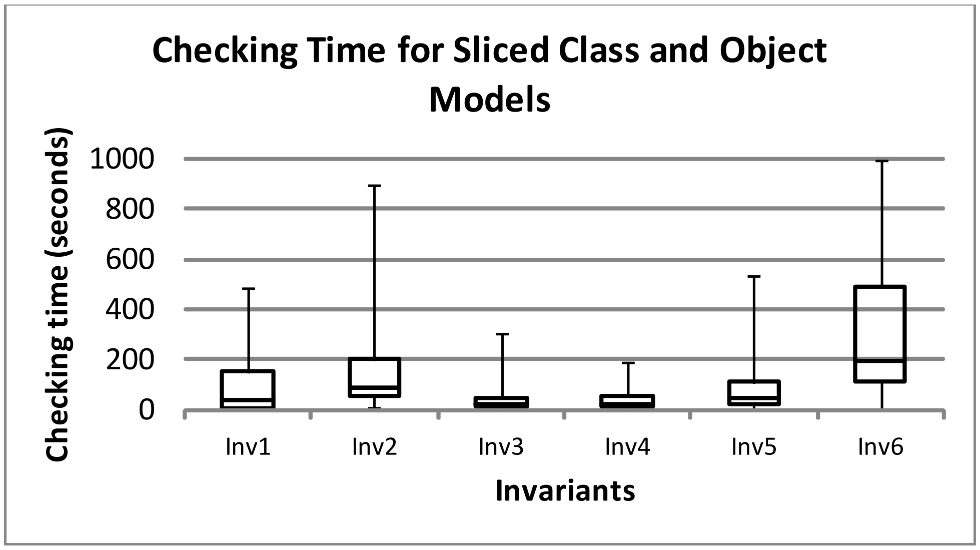


Figure 6.3. Box plot for the measurement of Checking Time for Sliced class and object Models (*CTSM*)

for each invariant. Note that to ensure the evaluation results are reliable, we calculated the CTS ten times for each pair of invariant and object model, and used its average value.

Figure 6.2 shows the distribution of the checking time for unsliced class and object models for each invariant. Although most of the *CTUM*s are less than 2000 seconds, the *CTUM* could achieve 7860 seconds (i.e., more than two hours) in the worst case scenario (see the

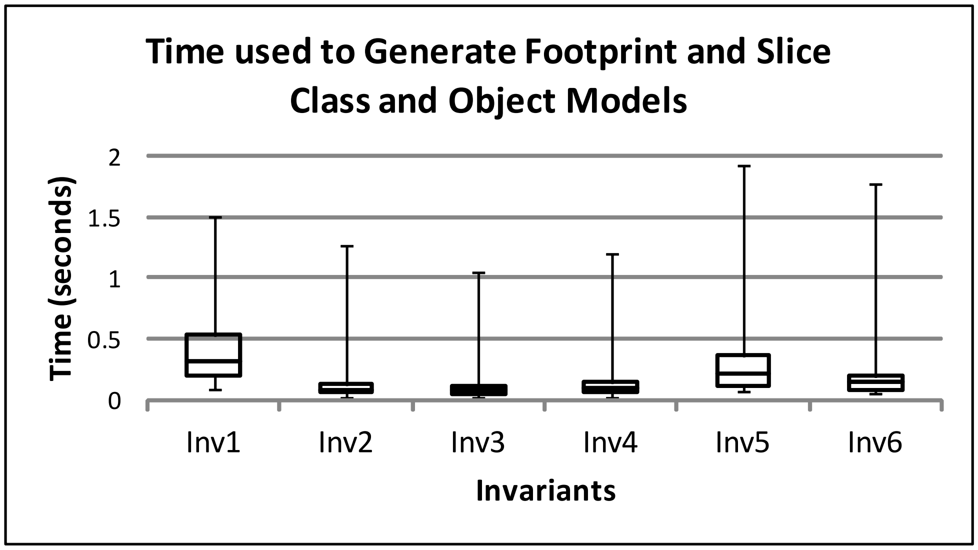


FIGURE 6.4. Box plot for the measurement of ST

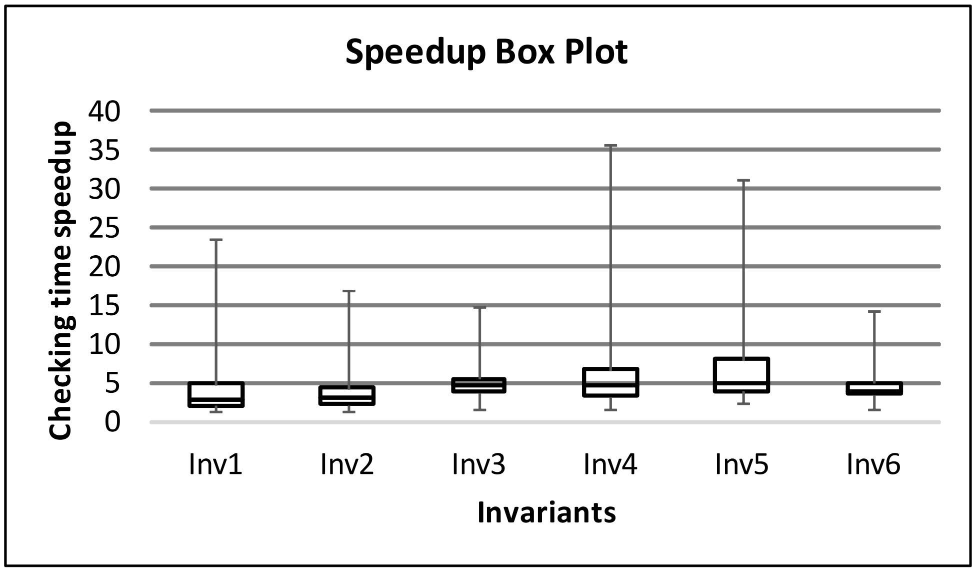


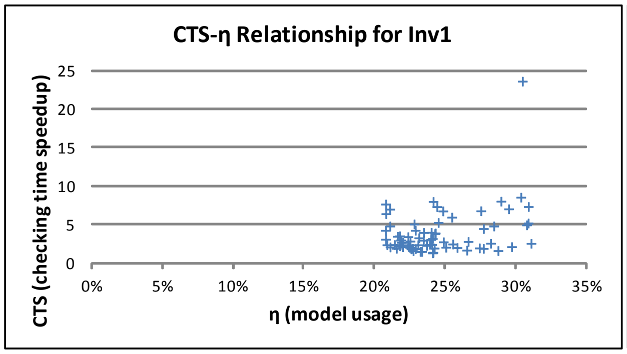
FIGURE 6.5. Box plot for the measurement of Checking Time Speedup (CTS)

box plot for *Inv6*). Figure 6.3 shows the distribution of the checking time for sliced class and object models for each invariant. All the *CTSM*s are less than 1000 seconds, and most of the *CTSM* are less than 200 seconds. Compared with *CTUM* and *CTSM*, the time used to generate footprints and to slice class and object models (*ST*) is quite small. Figure 6.4 shows the distribution of the *ST*s for each invariant. All the *ST*s are less than two seconds.

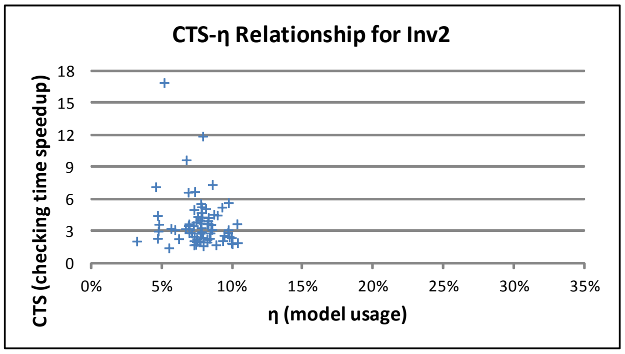
Figure 6.5 shows the distribution of the checking time speedup for each invariant. All the *CTS*s are calculated using Metric 1. Given *Inv5*, the *CTS*s for the 73 models vary from 2.5 (minimum CTS) to 31.1 (maximum CTS). The medium CTS for *Inv3* is about 5.0, the first quartile is about 4.0 and the third quartile is about 8.0. Since the *CTS* for each pair of invariant and object model is above 1, the value of $CTSM + ST$ must be smaller than *CTUM* (refer to Metric 1). Thus, the slicing technique can improve the efficiency of the invariant checking (refer to *RQ1*).

Figure 6.5 also shows how significantly the slicing technique improves the checking efficiency. In the worst case scenario, the *CTS* is close to 1.5 (see the minimum *CTS* for Inv1), while in the best case scenario, the *CTS* is close to 36.0 (see the maximum *CTS* for Inv4). Since the first quartile of *CTS*s for each invariant is above 2.0, the slicing technique can significantly improve the checking efficiency for three fourths of the object models used in the evaluation (i.e., 55 object models). In summary, the slicing technique can reduce the time used for invariant checking.

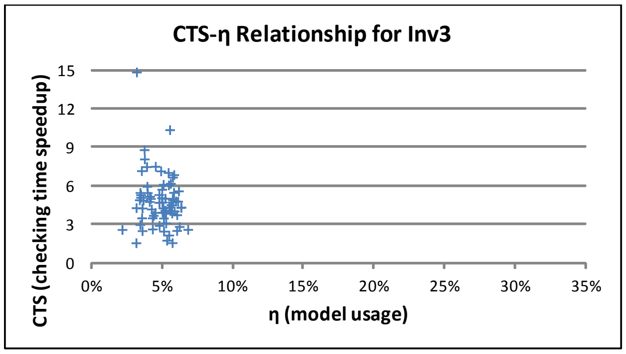
6.1.2.2. *Correctness of the Slicing Technique.* In the invariant checking, the checking tool matches an object with an invariant if the object is an instance of a class in the context of which the invariant is defined, and checks the object against its matched invariant. If an object satisfies its matched invariant, the object would be identified as a valid object w.r.t. the invariant. To analyze the correctness of the slicing technique, we need to identify both matched and valid objects for each invariant in unsliced and sliced object models respectively. Given an invariant, if (1) its matched objects in the unsliced object model are the same as the matched objects in the sliced object model, and (2) its valid objects in the unsliced object model are the same as the valid objects in the sliced object model, the slicing technique preserves the checking results for the invariant (refer to *RQ1.2*).



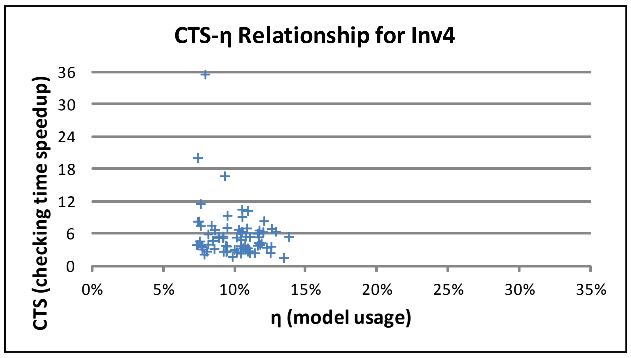
(A) CTS-η Relationship in the context of *Inv1*



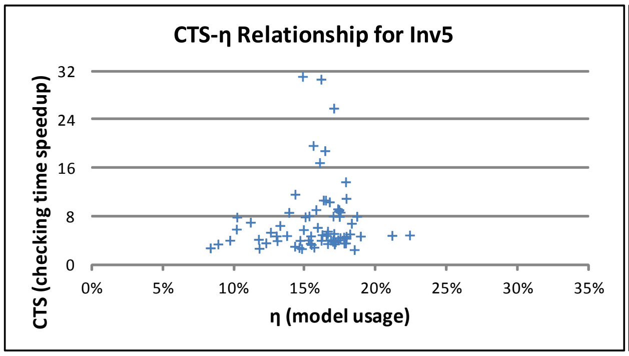
(B) CTS-η Relationship in the context of *Inv2*



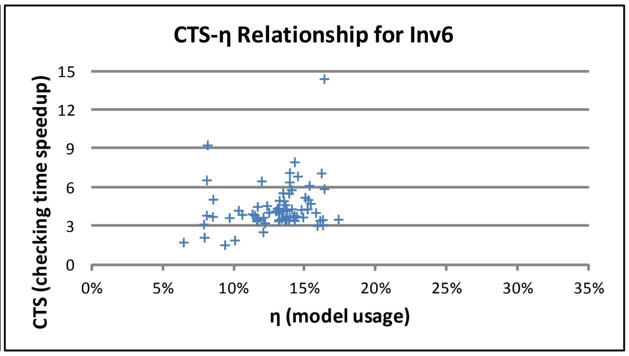
(C) CTS-η Relationship in the context of *Inv3*



(D) CTS-η Relationship in the context of *Inv4*



(E) CTS-η Relationship in the context of *Inv5*



(F) CTS-η Relationship in the context of *Inv6*

FIGURE 6.6. CTS-η Relationship in the context of each invariant

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

6.1.2.3. *Relationship between the Checking Time Speedup and the Model Usage.* The evaluation framework continues with the steps described in Section 6.1.2.1, and calculates the model usage (η) for each *CTS* based on Metric 2. In total, there are 73 ηs for each invariant. We drew a scatter plot of the η (i.e., X-axis) and *CTS* (i.e., Y-axis) for each invariant (see Figures 6.6a-6.6f).

The scatter plots show that when the model usages are small (i.e., less than 20%), most of the *CTS*s are above 3.0 (see Figure 6.6b, Figure 6.6c, Figure 6.6d, Figure 6.6e, and Figure 6.6f). The scatter plots also show that when the model usage increases (e.g., greater than 20%), almost half of the *CTS*s are below 3.0 (see Figure 6.6a) and 9% of the *CTS*s (e.g., about six *CTS*s) are below 2.0. In summary, the model usage has negative impact on the checking time speedup.

6.1.3. Threats to Validity.

The Java metamodel used in the evaluation may not be well representative of class models used in practice. It is possible that the slicing technique would be less efficient in other settings (e.g., the UML metamodel). To mitigate this threat, we also used the UML metamodel and its instances to evaluate the slicing technique. The evaluation results for the UML metamodel showed that the slicing technique can significantly improve the invariant checking while preserving the checking results.

Construction threats lie in the way we defined the formulas used in the evaluation. The choices of formula and statistical analysis may have impact on evaluation results and conclusions. For example, Metric 1 does not take the model loading time into consideration.

The reason we made this choice is because the model loading time is relatively small compared with the checking time (e.g., seconds v.s. minutes/hours). In addition, models may be already loaded when performing the invariant checking.

The validity of the evaluation results could be affected by calculations performed by the evaluation framework. To mitigate this threat, we calculated the *CTS* ten times for each pair of invariant and model, and used its average value. In addition, we used different sizes of object models (see Figure 6.1) in the evaluation to ensure the results are reliable.

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

6.2. Evaluating Contract-aware Slicing of Class Model

Q2 can be divided into the following research questions:

RQ2.1 Can the slicing technique improve the efficiency of the contact checking?

RQ2.2 Can the slicing technique preserve the contract checking results?

Metric 1 (*CTS*) given at the beginning of Section 6.1 is used to calculate the invariant checking time speedup achieved by the first type of the slicing technique. Unlike the first type of slicing technique, the second type of slicing technique is used for contract checking, and it may produce more than one model fragments from a class model. Since the first type of the slicing technique produces only one model fragment (i.e., sliced class model) from an input class model, Metric 1 cannot be used to estimate the effectiveness of the slicing technique used for contract checking. In addition, the contract checking 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 the contract checking time may vary w.r.t. the search scope. Therefore, to answer *RQ2.1*, we need to check whether the contract checking time for the entire class model is greater than the contract checking time for the corresponding model fragment 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).

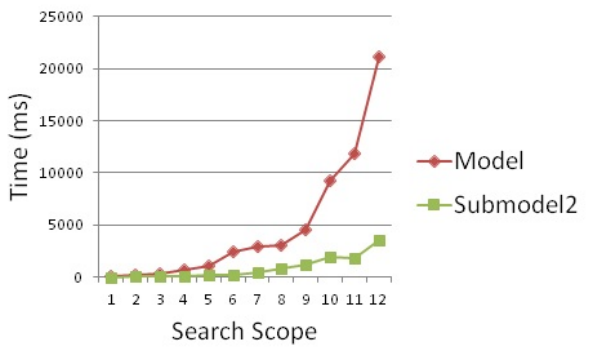
The slicing technique aims to improve the efficiency of the contract checking. Thus it should not change the checking results. To answer *RQ2.2*, we need to check whether the contract checking results for unsliced class models are the same as that for sliced class models.

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). We also describe the threats to validity we identified.

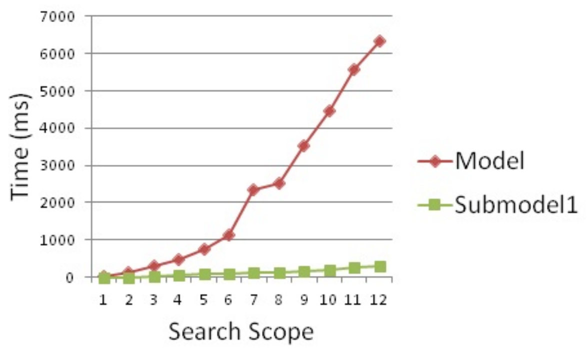
6.2.1. Evaluation Results for LRBAC.

We applied the slicing technique to the LRBAC class model in Figure 4.12, and it took 827 milliseconds to decompose the class model into 4 fragments as shown in Figure 4.16. The Alloy Analyzer was used to analyze the contracts that may violate the invariants given in Table 4.5 in the class model and the corresponding model fragment respectively.

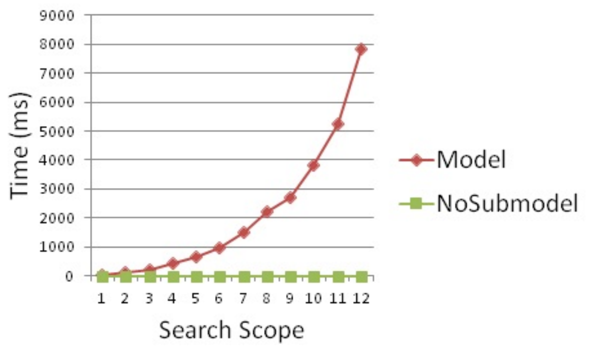
Figure 6.7 shows the results of an evaluation we performed on the entire $LRBAC$ class model and the corresponding model fragments. Each subfigure in 6.7 has an x axis, namely $SearchScope$, indicating the maximum number of instances the Alloy Analyzer can produce for a class, and a y axis, namely $Time$, showing the total analysis time (in millisecond) for building the SAT formula and finding an Alloy instance. For example, Figure 6.7a shows the time used to analyze the invariant $Inv1$ in the entire model $Model$ (see Figure 4.12) and the model fragment $Submodel2$ (see Figure 4.16) respectively.



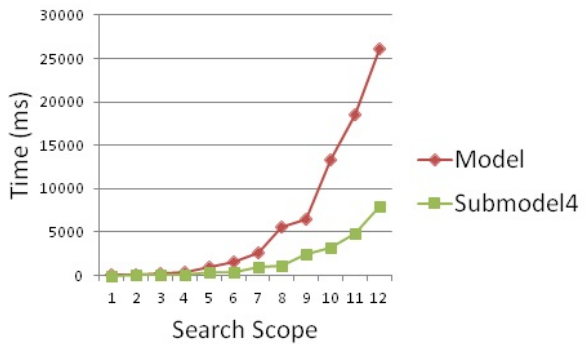
(A) Analyzing Inv1



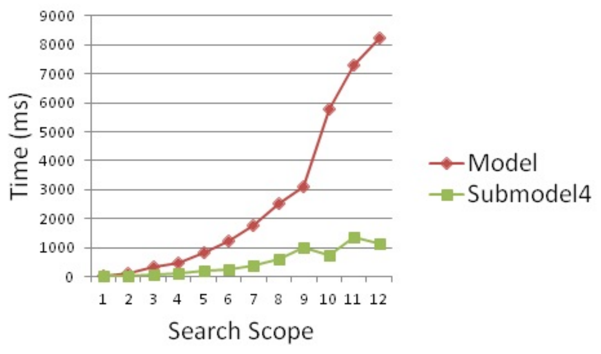
(B) Analyzing Inv2



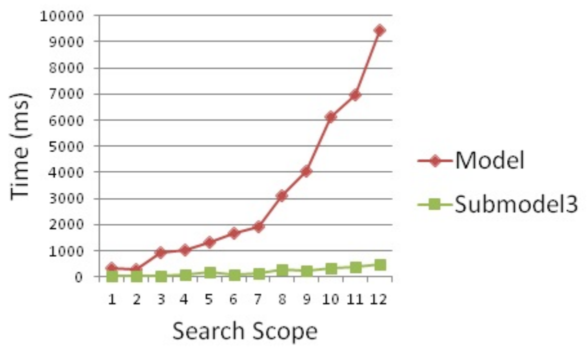
(C) Analyzing Inv3



(D) Analyzing Inv4



(E) Analyzing Inv5



(F) Analyzing Inv6

FIGURE 6.7. Checking the contracts in the entire LRBAC model and the corresponding model fragments w.r.t. each invariant given in Table 4.5

The difference between the time used for analyzing $Model$ and that for $Submodel2$ 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 $Submodel2$ is still below 5000 ms. Figure 6.7c shows that the analysis time for the $Inv3$ invariant remains at 0, and $Inv3$ is not included in any model fragment. This is because $Inv3$ was removed after the dependency analysis identified the invariant as an analysis-irrelevant element. In summary, the slicing technique can significantly reduce the time used for contract checking when the Alloy search scope is large (e.g., more than 10).

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 dissertation could speed up the verification process for large models.

The evaluation also showed that the contract checking results are preserved by the slicing technique (see *RQ2.2*). For example, the analysis performed on the unsliced model and the model fragments both found that the constraints specified in invariant $Inv2$, $Inv5$ and $Inv6$ were violated by operation $UpdateUserID$, $UpdateMaxRoles$ and $UpdateObjID$ respectively. The analysis of the full model also revealed that Inv3 is not violated by operation invocations; this is consistent with the identification of Inv3 as an analysis-irrelevant element.

Table 6.2 shows the contract checking results for the entire $LRBAC$ model and the corresponding model fragments. The first row in the table shows a list of Alloy search scopes, ranging from one to 12. The second row in the table shows the contract checking results for the *Inv1* invariant in the context of the entire $LRBAC$ model (*Model*). The results show that no Alloy instance was found for *Model* within the corresponding Alloy search scopes, indicating that the contracts defined in the $LRBAC$ model did not violate *Inv1*. Since *Inv1* is also included in the model fragment *Submodel*, *Inv1* was checked in the context of *Submodel*. The checking results for *Submodel* revealed (see the third row in the table) that the

Table 6.2. Contract checking results for the entire LRBAC model and corresponding model fragments

Search Scope	1	2	3	4	5	6	7	8	9	10	11	12
Instance Found in Model for Inv1	No	No	No	No	No	No	No	No	No	No	No	No
Instance Found in Submodel2 for Inv1	No	No	No	No	No	No	No	No	No	No	No	No
Instance Found in Model for Inv2	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Submodel1 for Inv2	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Model for Inv3	No	No	No	No	No	No	No	No	No	No	No	No
Inv3 is not included in any fragment												
Instance Found in Model for Inv4	No	No	No	No	No	No	No	No	No	No	No	No
Instance Found in Submodel4 for Inv4	No	No	No	No	No	No	No	No	No	No	No	No
Instance Found in Model for Inv5	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Submodel4 for Inv5	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Model for Inv6	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Submodel3 for Inv6	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

contracts included in *Submodel* did not violate *Inv1* within the corresponding Alloy scopes; this is consistent with the checking results for *Model.*

The fourth and fifth rows in Table 6.2 show that the checking results for *Inv2* in the context of *Model* are the same as that for *Inv2* in the context of *Submodel1.* The sixth row in

Table 6.3. 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

Table 6.2 show that the contracts defined in *Model* did not violate the *Inv3* invariant. The seventh row show that *Inv3* is not included in any model fragment, indicating that it is an analysis-irrelevant element. Therefore, it will not be violated by any operation invocation; this is consistent with the checking results for *Inv3* in the context of *Model*. The rest of the rows in Table 6.2 show that the checking results for *Inv4*, *Inv5*, and *Inv6* in the context of *Model* are the same as that in the context of the corresponding model fragments. In summary, the contract checking results are preserved by the slicing technique.

6.2.2. Evaluation Results for Other Class Models. We also used a variety of class models to evaluate the slicing technique. Below is a list of class models used in the evaluation:

- *CarRental*: The class model, from the USE project [17], describes the concepts of the car rental service;
- *Project*: The class model, from the USE Project [17], describes the concepts of a project management system;
- *CoachBus*: The class model, from the class model slicing paper [39], describes the information system of a bus company;
- *RoyalAndLoyal*: The class model, from the OCL book [47], describes loyalty programs for companies that offer their customers various kinds of bonuses.

Table 6.4. Slicing Results

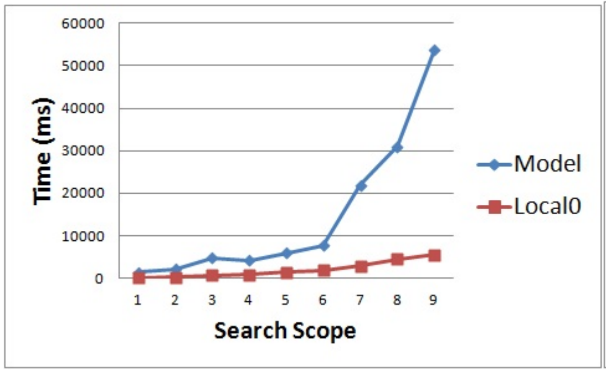
Model	**Slicing Time (ms)**	**Fragment**	**Size**	**# of Invs**	**# of Op Contracts**
CarRental	889	Local0	4	1	1
		Leftover1	9	1	1
Project	846	Leftover0	24	3	4
CoachBus	780	Local0	4	1	1
		Local1	4	1	1
		Leftover2	12	2	3
RoyalAndLoyal	837	Locl0	6	1	1
		Loftover1	32	7	2

Table 6.3 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, eight invariants and two 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 six operation contracts.

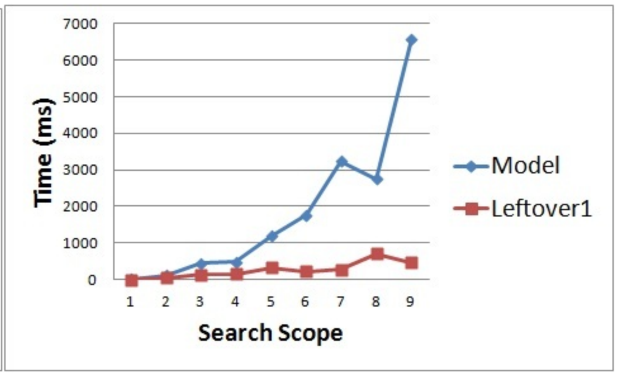
We applied the slicing technique to these class models, and Table 6.4 shows the slicing results. For example, it took 889 milliseconds to generate two fragments, namely *Local0* and *Leftover1*, from the *CarRental* class model. *Local0* has four elements, one invariant and one operation contract, and *Leftover1* has nine elements, one invariant and one operation contract. The Alloy Analyzer was used to analyze the contracts defined in the class model. Specifically for an invariant defined in a class model, the Alloy Analyzer checks the invariant in the context of the class model and the corresponding model fragment respectively.

In the remainder of this section we describe the evaluation results for each class model.

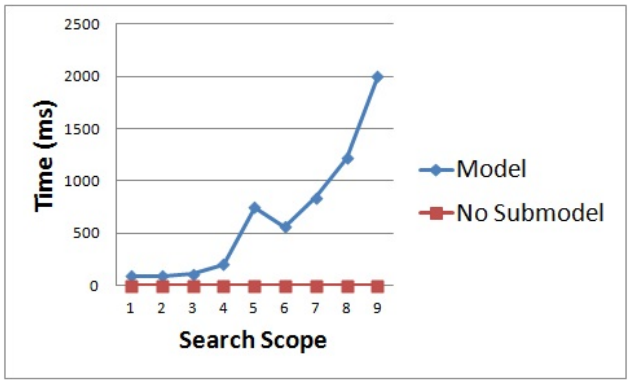
6.2.2.1. *Evaluation Results for the CarRental Class Model.* Figure 6.8 shows the results of an analysis we performed on the entire *CarRental* class model and the corresponding model



(A) Analyzing Invariant Person3



(B) Analyzing Invariant Branch2



(C) Analyzing Invariant Employee1

FIGURE 6.8. Analyzing the Invariants in the Entire *CarRental* Class Model and the Corresponding Model Fragments

fragments. Two fragments, namely *Local0* and *Leftover1* (see Table 6.4), 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 nine, while the time used for analyzing *Person3* in *Model* and the time used for analyzing *Branch2* in *Model* increases significantly when the Alloy search scope is greater than six. Note that there are nine invariants in the *CarRental* class model, while only two 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, Figure 6.8c shows that the *Employee1* invariant is not included in any model fragment, and its analysis time remains at 0, indicating that the invariant will not be violated by any operation invocation. The analysis of the entire model also revealed that the *Employee1* invariant is not violated by operation invocations; this is consistent with the identification of Employee1 as an analysis-irrelevant element. In summary, the slicing technique can significantly reduce the time used for contract checking in the context of the *CarRental* class model.

The evaluation also showed that the contract checking results are preserved by the slicing technique. Table 6.5 shows the contract checking results for the entire *CarRental* class model and the corresponding model fragments. The first row in the table shows a list of Alloy search scopes, ranging from one to nine. The second row in the table shows the contract checking results for the *Person3* invariant in the context of the entire *CarRental* model (*Model*). The results revealed that Alloy instances were found for *Model* when the Alloy search scope is greater than one, indicating that the contracts defined in the *CarRental* class model violated the *Person3* invariant. Since *Person3* is also included in the model fragment *Local0*, *Person3* was checked in the context of *Local0*. The checking results revealed (see the third row in the table) that the contracts included in *Local0* violated *Person3* when the Alloy search scope is greater than one; this is consistent with the checking results for the entire *CarRental* model.

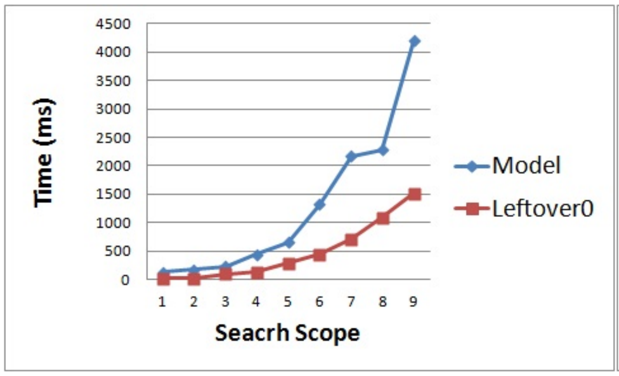
The fourth and fifth rows in Table 6.5 revealed that the checking results for *Branch2* in the context of *Model* are the same as that for *Branch2* in the context of *Submodel1*. The sixth row in Table 6.5 show that the contracts defined in *Model* did not violate the *Employee1* invariant. The seventh row show that *Employee1* is not included in any model fragment, indicating that it is an analysis-irrelevant element. Therefore, it will not be violated by any operation invocation; this is consistent with the checking results for *Employee1* in the context of *Model*. In summary, the contract checking results for the *CarRental* class model are preserved by the slicing technique.

Table 6.5. Contract checking results for the entire CarRental model and corresponding model fragments

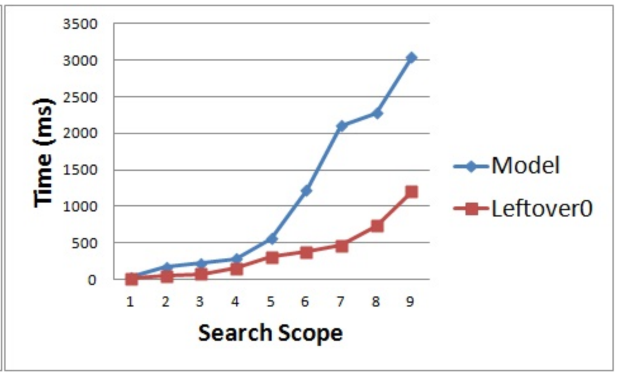
Search Scope	1	2	3	4	5	6	7	8	9
Instance Found in Model for Person3	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Local0 for Person3	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Model for Branch2	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Leftover1 for Branch2	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Model for Employee1	No	No	No	No	No	No	No	No	No
Employee1 is not included in any fragment									

6.2.2.2. *Evaluation Results for the Project Class Model.* Figure 6.9 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 6.4). The *Leftover0* fragment shares the same invariants and operation contracts with the entire class model. Since the model fragment (i.e., 24 elements) has fewer elements than the entire class model (i.e., 47 elements), it could be expected that the analysis time used for the model fragment is less than that for the entire class model. This is confirmed by the analysis results (see Figure 6.9).

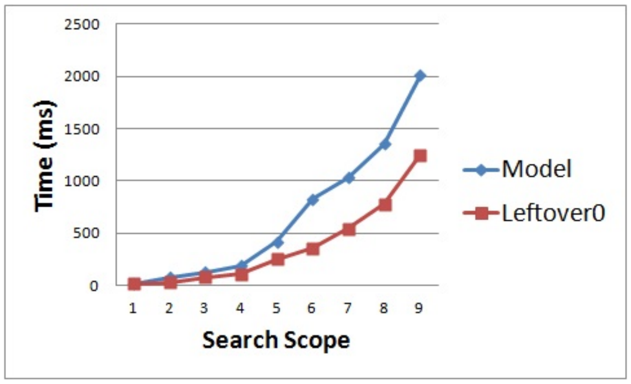
When the Alloy search scope is small (e.g., less than six), the difference between the analysis time used for the model fragment and the analysis time used for the entire class model is not significant. For example, when the Alloy search scope is five, the time used for analyzing *NotOverLoaded* in *Leftover0* is about 350 milliseconds, while the time used for analyzing *NotOverLoaded* in *Model* is about 550 milliseconds. When the Alloy search scope is large (e.g., equal to or more than six), the analysis time used for the entire class model is almost two times more than the time used for the model fragment. For example, when the



(A) Analyzing Invariant EmployeesInProject



(B) Analyzing Invariant NotOverloaded



(C) Analyzing Invariant ActiveProject

FIGURE 6.9. Analyzing the Invariants in the Entire Project Class Model and the Corresponding Model Fragments

Alloy search scope is six, the time used for analyzing *NotOverLoaded* in *Leftover0* is about 450 milliseconds, while the time used for analyzing *NotOverLoaded* in *Model* is about 1200 milliseconds. In summary, the slicing technique can significantly reduce the time used for contract checking in the context of the *Project* class model.

The evaluation also showed that the contract checking results are preserved by the slicing technique. Table 6.6 shows the contract checking results for the entire *Project* class model and the corresponding model fragments. The first row in the table shows a list of Alloy search scopes, ranging from one to nine. The second row in the table shows the contract checking results for the *EmployeesInProject* invariant in the context of the entire *Project*

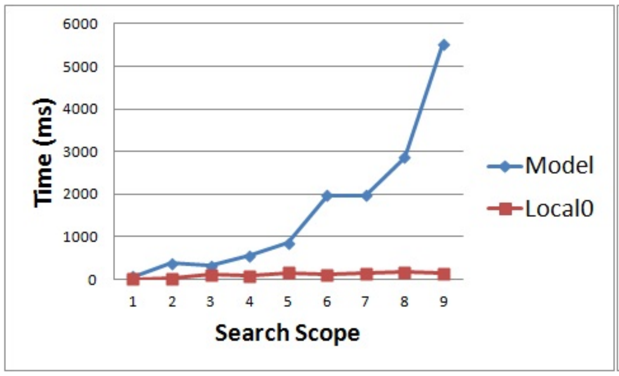
Table 6.6. Contract checking results for the entire Project model and corresponding model fragments

Search Scope	1	2	3	4	5	6	7	8	9
Instance Found in Model for EmployeesInProject	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Leftover0 for EmployeesInProject	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Model for NotOverLoaded	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Leftover0 for NotOverLoaded	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Model for ActiveProject	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Leftover0 for ActiveProject	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

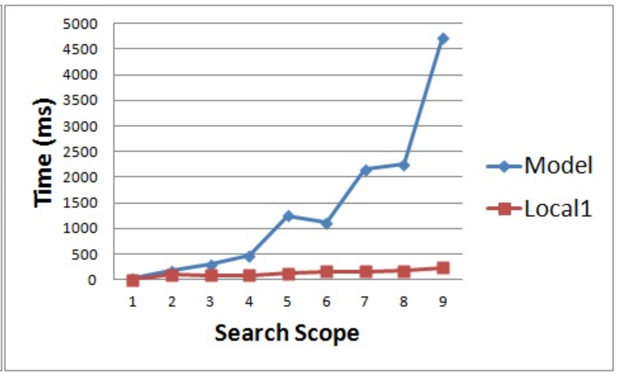
model (*Model*). The results revealed that Alloy instances were found for *Model* when the Alloy search scope is greater than one, indicating that the contracts defined in the *Project* class model violated the *EmployeesInProject* invariant. Since *EmployeesInProject* is also included in the model fragment *Leftover0*, *EmployeesInProject* was checked in the context of *Leftover0*. The checking results revealed (see the third row in the table) that the contracts included in *Leftover0* violated *EmployeesInProject* when the Alloy search scope is greater than one; this is consistent with the checking results for the entire *Project* model.

The fourth and fifth rows in Table 6.6 revealed that the checking results for *NotOverLoaded* in the context of *Model* are the same as that for *NotOverLoaded* in the context of *Leftover0*. The sixth and seventh rows in Table 6.6 revealed that the checking results for *ActiveProject* in the context of *Model* are the same as that for *ActiveProject* in the context of *Leftover0*. In summary, the contract checking results for the *Project* class model are preserved by the slicing technique.

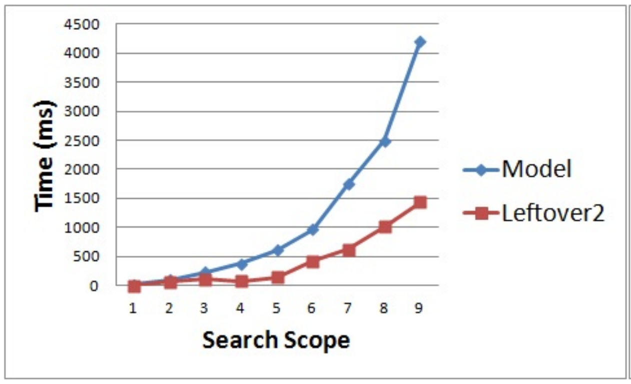
6.2.2.3. *Evaluation Results for the CoachBus Class Model.* Figure 6.10 shows the results of an analysis we performed on the entire *CoachBus* class model and the corresponding



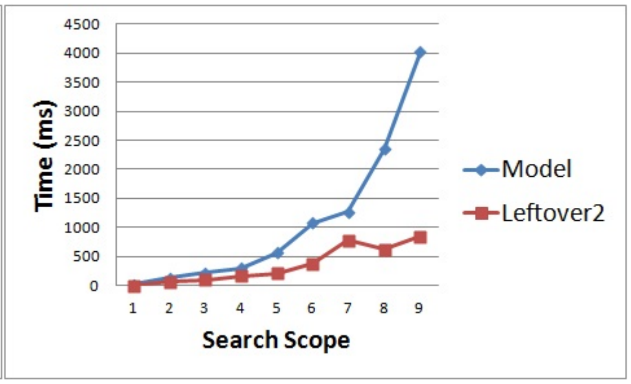
(A) Analyzing Invariant NonNegativeAge



(B) Analyzing Invariant UniqueT-icketNumber



(C) Analyzing Invariant MinCoachSize



(D) Analyzing Invariant MaxCoachSize

Figure 6.10. Analyzing the Invariants in the Entire CoachBus Class Model and the Corresponding Model Fragments

model fragments. Three model fragments, namely *Local0*, *Local1* and *Leftover2* (see Table 6.4), 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 nine. The time used for analyzing *MinCoachSize* in *Leftover2* steadily grows as the Alloy search scope increases. But it is still less than the time used for analyzing *MinCoachSize* in the entire *CoachBus* class model. The time used for analyzing *MaxCoachSize* in *Leftover2* is much less than the time used for analyzing *MaxCoachSize* in *Model* when the Alloy search scope is large (greater than seven). Note that the *CoachBus* class model has six invariants

Table 6.7. Contract checking results for the entire CoachBus model and corresponding model fragments

Search Scope	1	2	3	4	5	6	7	8	9
Instance Found in Model for NonNegativeAge	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Local0 for NonNegativeAge	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Model for UniqueTicketNumber	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Local1 for UniqueTicketNumber	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Model for MinCoachSize	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Leftover2 for MinCoachSize	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Model for MaxCoachSize	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Leftover2 for MaxCoachSize	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

(see Table 6.3), and two of them are not shown in Figure 6.10 because they were removed as analysis-irrelevant elements. In summary, the slicing technique can significantly reduce the time used for contract checking in the context of the *CoachBus* class model.

The evaluation also showed that the contract checking results are preserved by the slicing technique. Table 6.7 shows the contract checking results for the entire *CoachBus* class model and the corresponding model fragments. The first row in the table shows a list of Alloy search scopes, ranging from one to nine. The second row in the table shows the contract checking results for the *NonNegativeAge* invariant in the context of the entire *CoachBus* model (*Model*). The results revealed that Alloy instances were found for *Model* when the Alloy search scope is greater than one, indicating that the contracts defined in the *CoachBus* class model violated the *NonNegativeAge* invariant. Since *NonNegativeAge* is also included in the model fragment *Local0*, *NonNegativeAge* was checked in the context of *Local0*. The checking results revealed (see the third row in the table) that the contracts included in *Local0* violated *NonNegativeAge* when the Alloy search scope is greater than one; this is consistent with the checking results for the entire *CoachBus* model.

The fourth and fifth rows in Table 6.7 revealed that the checking results for *UniqueT-icketNumber* in the context of *Model* are the same as that for *UniqueTicketNumber* in the context of *Local1.* The sixth and seventh rows in Table 6.7 revealed that the checking results for *MinCoachSize* in the context of *Model* are the same as that for *MinCoachSize* in the context of *Leftover2.* The eighth and ninth rows in Table 6.7 revealed that the checking results for *MaxCoachSize* in the context of *Model* are the same as that for *MaxCoachSize* in the context of *Leftover2.* In summary, the contract checking results for the *CoachBus* class model are preserved by the slicing technique.

6.2.2.4. *Evaluation Results for the RoyalAndLoyal Class Model.* Figure 6.11 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 6.4), were generated from the *RoyalAndLoyal* class model. The *Local0* model fragment has only one invariant (*UniqueName*) and one operation contract. Compared with the time used for analyzing *UniqueName* in the entire class model (up to 18000 milliseconds), the time used for analyzing *UniqueName* in *Local0* is significantly small (less than 100 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 nine. The time used for analyzing *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 about 24000 milliseconds. Note that the *RoyalAndLoyal* class model has nine invariants (see Table 6.3), and only one of them is not shown in Figure 6.11 because it was

TABLE 6.8. Contract checking results for the entire RoyalAndLoyal model and corresponding model fragments

Search Scope	1	2	3	4	5	6	7	8	9
Instance Found in Model for UniqueName	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Local0 for UniqueName	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Model for ProgramPartner1	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Leftover1 for ProgramPartner1	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Model for nrOfParticipants	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Leftover1 for nrOfParticipants	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Model for ServiceLevel1	No	No	No	No	No	No	No	No	No
Instance Found in Leftover1 for ServiceLevel1	No	No	No	No	No	No	No	No	No
Instance Found in Model for Customer10	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Leftover1 for Customer10	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Model for sizesAgree	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Leftover1 for sizesAgree	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Model for Customer1	No	No	No	No	No	No	No	No	No
Instance Found in Leftover1 for Customer1	No	No	No	No	No	No	No	No	No
Instance Found in Model for CustomerCard3	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes
Instance Found in Leftover1 for CustomerCard3	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes

removed as analysis-irrelevant element. In summary, the slicing technique can significantly reduce the time used for contract checking in the context of the *RoyalAndLoyal* class model.

The evaluation also showed that the contract checking results are preserved by the slicing technique. Table 6.8 shows the contract checking results for the entire *RoyalAndLoyal* class model and the corresponding model fragments. The first row in the table shows a list of Alloy search scopes, ranging from one to nine. The second row in the table shows the contract checking results for the *UniqueName* invariant in the context of the entire *RoyalAndLoyal* model (*Model*). The results revealed that Alloy instances were found for *Model* when the Alloy search scope is greater than one, indicating that the contracts defined in the *RoyalAndLoyal* class model violated the *UniqueName* invariant. Since *UniqueName* is also included in the model fragment *Local0*, *UniqueName* was checked in the context of *Local0*. The checking results revealed (see the third row in the table) that the contracts included in *Local0* violated *UniqueName* when the Alloy search scope is greater than one; this is consistent with the checking results for the entire *RoyalAndLoyal* model.

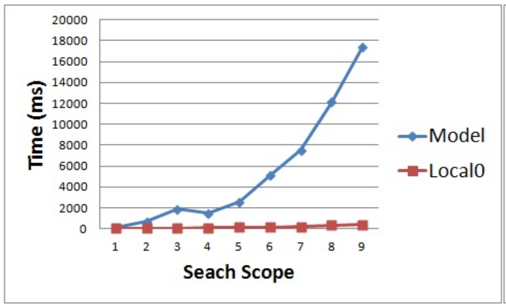
The rest of the rows in Table 6.8 show that the checking results for *ProgramPartner1*, *nrOfParticipants*, *ServiceLevel1*, *Customer10*, *sizesAgree*, *Customer1* and *CustomerCard3* in the context of *Model* are the same as that in the context of the corresponding model fragments. In summary, the contract checking results for the *RoyalAndLoyal* class model are preserved by the slicing technique.

6.2.3. Threats to Validity.

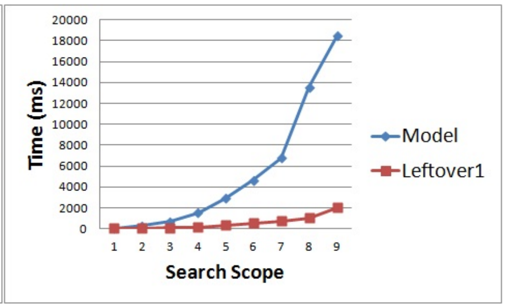
The validity of the evaluation results could be affected by calculations performed by the evaluation framework. To mitigate this threat, we calculated the contract checking time ten times for each pair of invariant and class model, and used its average value. In addition, we used different sizes of class models in the evaluation to ensure the results are reliable.

Construction threats lie in the way we defined the formulas used in the evaluation. The choices of formula and statistical analysis may have impact on evaluation results and conclusions. For example, the class model slicing is not taken into consideration when we compare the checking time used for the model fragments and for the entire class model. This is mainly because the slicing time is relatively small compared with the contract checking time when the Alloy search scope is large (e.g., less than one second v.s. seconds). Moreover, the slicing technique is used only once for each class model in the evaluation, and it may produce more than one model fragments. On the other hand, the contract checking may be performed more than once on a model fragment. Therefore, it may introduce bias when we simply compare the checking time used for the entire class model with the checking time used for one model fragment plus the class model slicing time.

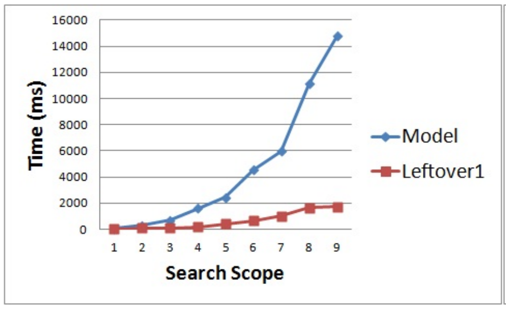
In addition, the model loading time is not taken into consideration in the evaluation. The reason we made this choice is because (1) the class model loading time is relatively small compared with the contract checking time (e.g., milliseconds v.s. seconds), and (2) class models may be already loaded when performing the contract checking.



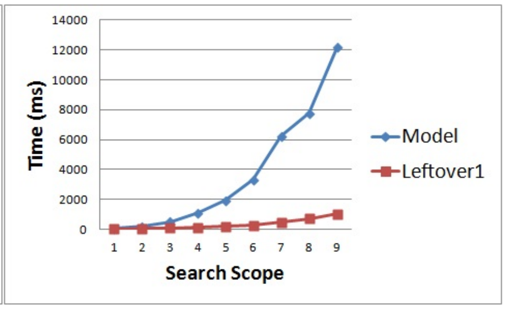
(A) Analyzing Invariant UniqueName



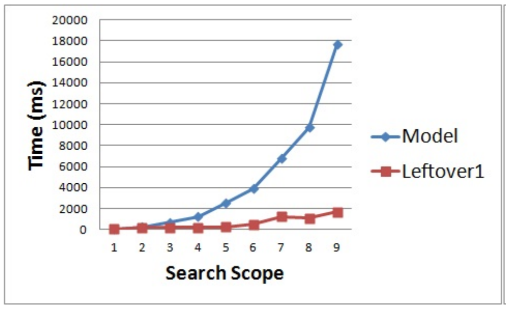
(B) Analyzing Invariant ProgramPartner1



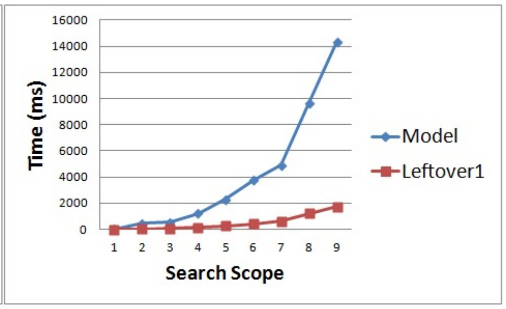
(C) Analyzing Invariant nrOfParticipants



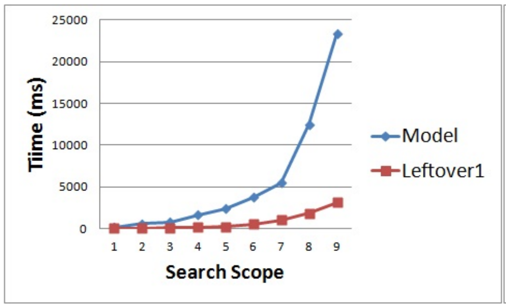
(D) Analyzing Invariant ServiceLevel1



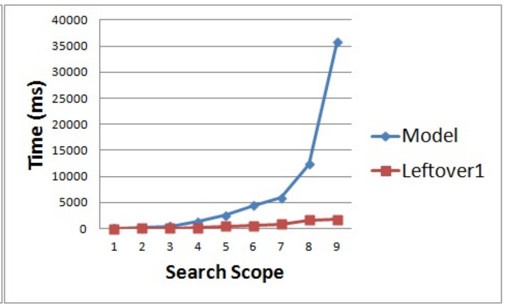
(E) Analyzing Invariant Customer10



(F) Analyzing Invariant sizesAgree



(G) Analyzing Invariant Customer1



(H) Analyzing Invariant CustomerCard3

FIGURE 6.11. 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) and provide an overview of plans for future related work (Section 7.2).

7.1. Contribution

The contribution of this dissertation includes (1) a state-of-the-art survey on class model slicing techniques, (2) a rigorous technique that supports co-slicing of class models and object configurations, (3) a rigorous technique that supports slicing of class models including both invariants and operation contracts, (4) a research prototype that provides implementations of two proposed class model slicing techniques, and (5) an evaluation framework for the proposed slicing techniques.

For the state-of-the-art survey, we conducted a systematic literature review on class model slicing techniques. The systematic review compared current class model slicing techniques and identified their limitations through a systematic evaluation. The review follows a carefully designed paper selection procedure, and identified eight class model slicing techniques in scientific journals and conferences from 2003 to 2014.

We proposed a slicing technique to improve the efficiency of model analysis (i.e., *Invariant Checking*) that involves checking whether an instance of a class model satisfies the invariants defined in the class model. The slicing technique uses the invariant being checked to produce sliced class and object models from the input class and object models. The output of the slicing technique (i.e., sliced class and object models) can be more efficiently analyzed by invariant checking tools. Note that the slicing technique is not intended to improve the existing invariant checking algorithms. Instead, the technique aims to reduce the size of the checking inputs to make the analysis more efficient. It means our slicing technique preprocesses the input of the invariant checking process. Therefore the technique described in the dissertation is agnostic to the technological space. We applied the slicing technique to a variety of tools (e.g., Kermeta Workbench [14], USE [17] and Alloy Analyzer [18]), showing that the slicing technique is agnostic to the checking technologies the software developers are working with.

We also presented a slicing technique for class models that includes invariants and operation contracts. The slicing technique is used to improve the efficiency of a class model analysis technique (i.e., *Contract Checking*) that involves checking a sequence of operation invocations to uncover violations in specified invariants. The slicing technique can reduce the problem of analyzing a large model with many invariants 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 approach 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 (invariants and operation contracts) included in a generated slicing criterion.

We developed a research prototype that provides implementations of two proposed class model slicing techniques. The prototype was developed using Java language and Eclipse development platform. The UML class models used for the prototype are expressed using the Ecore standard (i.e., the de-facto standard to define class models), the OCL invariants and operation contracts are specified in textual files, and the object configurations are expressed using the XMI standard (i.e., the de-facto standard to serialize models). The prototype uses the Eclipse Modeling Framework (EMF) [44] to parse Ecore class models and XMI object configurations. Even though the evaluation framework builds upon Java and Eclipse, the slicing technique is not bound to a particular technical space, and it can be implemented using any language and framework.

We also developed an evaluation framework for the proposed slicing techniques. The evaluation framework builds upon both the EMF and the Eclipse OCL project [46]. The framework uses the EMF to parse and serialize class and object models. The framework uses the Eclipse OCL project to (1) parse the OCL invariants defined in a class model and (2) check an object model against the class model with OCL invariants. The evaluation framework also builds upon the Alloy Analyzer [19], and provides a transformation between class models and Alloy models. The generated Alloy models can be fed into the Alloy Analyzer for rigorous analysis of invariants and operation contracts defined in the class models.

The purpose of the evaluation is to check whether (1) the proposed slicing technique improves the efficiency of the invariant checking and the contract checking, and (2) the checking results for the sliced models are the same as the unsliced models. We have evaluated the slicing technique for invariant checking with the Java class model and 73 object models produced from the Eclipse plugins. The evaluation we performed provides evidence that the proposed slicing technique can significantly reduce the time (e.g., achieving checking speedup ranging from 1.5 to 36) to perform the invariant checking for the Java class model and its instances while preserving the checking results.

We have also evaluated the slicing technique for contract checking using a variety of class models. The results of the evaluation we performed showed that the proposed slicing technique can significantly reduce the time to perform the contract checking for large Alloy search scope. The evaluation also revealed that the proposed slicing technique can preserve the contract checking results.

7.2. Future Work

The proposed slicing techniques can be limited in their abilities to produce smaller models from a large class model. There may be constraints (operation contracts or invariants) 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 techniques described in this dissertation do not ensure that more than one class model elements will be removed from a class model. Therefore, one objective of our future work is to address the limitation of the proposed 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 models from the class model.

One of the proposed slicing techniques described in the dissertation is used to improve the efficiency of the invariant checking. However, the invariant checking is not the only usage scenario for the slicing technique. The slicing technique can be further improved and used for other modeling tasks that involve both class and object models. One future direction of this work would be to evaluate the impact of the slicing technique on other kinds of modeling tasks (e.g., model transformation).

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

Bibliography

[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] Xavier Blanc, Isabelle Mounier, Alix Mougenot, and Tom Mens. Detecting model inconsistency through operation-based model construction. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 511–520. IEEE, 2008.

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

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

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

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

[11] 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.

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

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

[14] 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.

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

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

[17] 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.

[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] S. Maoz, O. Ringert, and B. Rumpe. Semantically configurable consistency analysis for class and object diagrams. In *Model Driven Engineering Languages and Systems*, pages 153–167. Springer, 2011.

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

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

[35] 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.

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

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

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

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

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

[45] 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.

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

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

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

[49] 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.

[50] 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.

[51] 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.

[52] 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.

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