DISSERTATION PROPOSAL

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

Fall 2013

COLORADO STATE UNIVERSITY

October, 2013

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

Committee on Graduate Work

_____
Committee Member

_____
Committee Member

_____
Adviser

_____
Co-Adviser

_____
Department Head

ABSTRACT OF DISSERTATION PROPOSAL

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 proposal we describe our planned work on developing class model slicing techniques that take into consideration constraints expressed in the Object Constraint Language (OCL). The goal of the research is to develop techniques that 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 the consistency 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. A preliminary evaluation

we performed provides evidence that the proposed slicing techniques can significantly reduce the time to perform the analysis. We plan to further evaluate the proposed techniques to assess their performance and scalability.

Wuliang Sun
Department of Computer Science
Colorado State University
Fort Collins, CO 80523
Fall 2013

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

1.1 Motivation

In the model-driven development (MDD) area, models are the major artifacts that drive the software development process. Rigorous analysis of models can enhance the ability of developers to understand the system under development and to identify potentially costly design problems earlier. Class models expressed using the Unified Modeling Language (UML) [14] 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 [16][35][19][18][13][21][30]) for UML class models. However, given that the complexity of software systems is increasing, one can expect that the size of class models used to represent these complex systems will also grow significantly in size. The scalability of current model analysis tools will become an issue in these situations, and thus there is a need for techniques that support scalable rigorous analysis of UML class models.

Slicing techniques [47] produce reduced forms of artifacts that can be used to support, for example, analysis of artifact properties. Slicing techniques have been proposed for different software artifacts, including programs (e.g., see [15][47]), and models (e.g., see [3][6][12][22][24]). In the MDD area, model slicing techniques have been used to support a variety of modeling tasks, including model comprehension [3][6][24], analysis

[20][27][28], and verification [12][38][41].

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

Rigorous analysis of invariants and operation contracts expressed in the Object Constraint Language (OCL) [42] can be expensive when the class models are large. Model slicing techniques can be used in these situations to reduce large models to just those fragments that can be analyzed separately. This reduction can help reduce the cost of analysis. However, many of the existing class model slicing techniques do not take constraints expressed in auxiliary constraint languages into consideration when producing model slices. Their applicability is thus limited to situations in which the determination of slices does not require information found in constraints. This limits the utility of class model slicing techniques in (1) situations where invariants are needed to specify additional properties that cannot be expressed directly in a class model (e.g., acyclicity property), and (2) model-based software development approaches that are contract based (e.g., design by contract [29]).

1.2 Proposed Research

The proposed research is concerned with developing 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 will develop the slicing

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

As stated above we propose to develop a slicing technique to improve the efficiency of a model analysis technique that checks the consistency between an object configuration and a class model with invariants. Such analysis plays an important role in MDD in that it can help developers to uncover structural errors in the design class models developed in the early stages of software development. Existing techniques, however, are likely to perform the analysis using the entire model and full object configurations, and their scalability becomes an issue when the sizes of the model and object configurations are large. The proposed slicing technique takes as input a class model with an invariant and an object configuration conforming to the class model. It analyzes the dependency between the invariant and the class model, and extracts a model fragment consisting of elements that are only referenced by the invariant. The model fragment is used to extract an object configuration fragment from the input object configuration. The object configuration fragment is an instance of the model fragment. The output of the slicing technique (i.e., the extracted model/object configuration fragment) and the invariant, can be fed into an analysis tool (e.g., Kermeta [13]) for consistency check.

We are also developing a slicing technique to improve the efficiency of a model analysis technique we developed to check that operation contracts do not allow invariant violations when sequences of conforming operations are invoked [44]. The slicing technique is used to reduce the problem of analyzing a large model with many OCL constraints (including OCL operation contracts) to smaller subproblems that involve analyzing a model fragment against a subset of invariants and operation contracts. Each model fragment can be analyzed independently of other fragments. Given a class model

with OCL constraints, the slicing technique automatically generates slicing criteria consisting of a subset of invariants and operation contracts, and uses the criteria to extract model fragments. Each model fragment is obtained by identifying and analyzing relationships between model elements and the constraints included in a generated slicing criterion.

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

1.3 Expected Contributions

The major contribution of the research will be a model slicing platform that provides implementations of two model slicing techniques we will develop. Below is a list of expected contributions of the research:

1. A state-of-the-art survey on class model slicing techniques;
2. A rigorous technique that supports co-slicing of UML class models and object configurations;
3. A rigorous technique that supports slicing of UML class models including both OCL invariants and operation contracts;
4. A platform that provides implementations of two proposed class model slicing techniques;
5. An evaluation and validation framework for the proposed slicing techniques.

1.4 Proposal Structure

The rest of the proposal 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 proposed class model slicing techniques, and Chapter 5 presents the results of a preliminary evaluation of the slicing techniques. Chapter 6 discusses limitations of the proposed slicing techniques and the preliminary evaluation. Chapter 7 summarizes what has been done and what needs to be done for the proposed research work using a planned time line.

Chapter 2

Literature Review

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

Kitchenham et al. [23] described a systematic procedure to identifying and analyzing available literature relevant to a specific research topic. Tao [52] 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 [52]. 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 proposal 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 proposal 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 [14] and can have invariants and operation contracts expressed using the OCL [42]. Slicing techniques that take as inputs multiple models are also considered in the review if the inputs of the slicing techniques include class models.

2.2 Research Questions

The systematic review aims to answer the following research questions:

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

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

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

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

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

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

2.3 Search Strategy

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

2.3.1 Electronic and Manual Search

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

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

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

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

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

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

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

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

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

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

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

2.3.2 Exclusion/Inclusion Procedure

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

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

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

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

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

sion (i.e., accept or reject the paper) was not able to be made from the title and abstract of the paper, we further checked the paper's introduction and conclusion. In summary, 12 papers from the search result (i.e., [4][5][6][7] [20][22][26][27][37][38][39][40]) 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][41]) 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, [38][39][40][41], describe the same slicing technique, and thus were counted as one primary study. In the remainder of this section we describe these eight primary studies on UML class model slicing:

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

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

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

3. Sen et al. [37] proposed a slicing technique for metamodel pruning (e.g., removing unnecessary classes and properties from a metamodel). The slicing technique

takes as input a large metamodel and a slicing criterion including a set of classes and properties of interest, and produces a pruned metamodel that is a subset of the input metamodel. The pruned metamodel contains all the model elements specified in the slicing criterion and their dependent elements from the input metamodel. The dependency relationship between two elements is determined by a set of rules given in their paper. For example, one of their rules specifies that class A depends on class B if A is a subclass of B. The pruned metamodel also satisfies the structural constraints imposed by the input metamodel. Thus any instance of the pruned metamodel is an instance of the input metamodel. Their technique is domain independent and therefore can be used for slicing any metamodels that conform to the UML standard.

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

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

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

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

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

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

8. Shaiky et al. [38][39][40][41] used a slicing technique to improve the scalability of an analysis that involves checking if a UML class model has a valid instance that satisfies the invariants defined in the class model. The slicing technique reduces the analysis problem of checking a large class model with OCL invariants into smaller subproblems, where each subproblem involves checking a model fragment with a subset of OCL invariants. Each model fragment is part of the input class model and can be analyzed separately. The OCL invariants are used as slicing criteria in their technique, and model elements that are not referenced by any invariants are removed from the input class model. The model decomposition process is guided by the following rule: All constraints restricting the same model element should be checked together and therefore must be contained in the same model fragment. Their slicing technique can handle only UML class models including OCL invariants.

Table 2.4: An Evaluation of Class Model Slicing Techniques

No.	Paper	PS	CMT	SC	IM	AT
1	[22]	Model maintenance	Basic	Class model elements	No	Automatable
2	[4][5]	Metamodel management	Basic	Classes	No	Automated
3	[37]	Metamodel pruning	Basic	Classes and properties	No	Automated
4	[6][7]	Building model slicer	Basic, Inv and Op	Class model elements	No	Automated
5	[25][26]	Model comprehension etc.	Basic	Classes, scenarios, and model data	Dependency graph	Automated
6	[20]	Change impact analysis	Op	Operation contracts	Metamodel footprint	Automated
7	[27][28]	Comprehension and analysis	InvOp	Operation contracts	No	Automatable
8	[38][39] [40][41]	Analysis	Inv	Invariants	Dependency graph	Automated

2.5 Evaluation

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

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

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

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

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

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

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

2.6 Open Issues

One open issue regarding the existing class model slicing techniques is that many of them do not take invariants and operation contracts expressed in auxiliary constraint languages (e.g., OCL) into consideration when producing model slices. Their applicability is thus limited to situations in which the determination of slices does not require information found in constraints. This limits the utility of class model slicing techniques in (1) situations where invariants are needed to specify additional properties that cannot be expressed directly in a class model (e.g., acyclicity property), and (2) model-based software development approaches that are contract based (e.g., design by contract [29]). 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 proposal we focus on two types of class model analysis: one that involves checking the consistency between an object configuration and a class model with specified invariants, and the other that analyzes sequences of operation invocations to uncover invariant violations. Section 3.1 describes a list of state-of-the-art model consistency checkers, USE [16][35], Alloy [19][18], and Kermeta [13][21][30], that can be used to handle the first type of analysis we are interested in. Section 3.2 describes an approach we are developing to tackle the second type of class model analysis.

3.1 State-of-the-art Model Consistency Checkers

The UML Specification Environment (USE) [16][35] is a modeling tool developed by the Database Systems Group at Bremen University. The USE tool can be used to specify object-oriented systems. It allows developers to analyze a system by generating snapshots (i.e., system states) against user-specified properties (i.e., invariants). USE can be used to specify a system in the form of a UML class model with a set of invariants expressed using the OCL. The shell commands, provided by the USE tool, is used to create a system state in the form of an object diagram. The object diagram consists of linked objects, where each object is an instance of a class in the class model. The feedback provided by the USE analysis mechanism will highlight the invariant that is inconsistent

with the given object diagram when the object diagram does not conform to the class model.

Alloy [19][18] is a formal specification language that was developed by the Software Design Group at MIT. It has very good tool support in the form of the Alloy Analyzer that translates an Alloy specification into a boolean formula that is evaluated by embedded SAT-solvers. The Alloy Analyzer generates examples or counter-examples of certain properties by exploring a search space. The search space is typically bound by users in the form of limits on the number of entities to be included in the search space. An Alloy model consists of signature declarations, fields, facts and predicates. Each field belongs to a signature and represents a relation between two or more signatures. Facts are statements that define constraints on the elements of the model. Predicates are parameterized constraints that can be invoked from within facts or other predicates.

The Alloy Analyzer can be used to check the consistency between an object configuration and a class model with invariants. For example, class models can be specified using Alloy signatures and fields, invariants defined on class models can be specified using Alloy facts, and object configurations can be specified using Alloy predicates. If the Alloy Analyzer cannot return an instance of an Alloy model for a predicate specifying an object configuration within a bounded scope, an object configuration specified using the predicate may not be consistent with a class model expressed using the Alloy model.

The Kermeta language [13][21][30] was developed by Triskell Team at INRIA. It is an executable metamodeling language implemented on top of the Eclipse Modeling Framework (EMF) [43] within the Eclipse development environment. It has been used for specifying metamodels, models, and model transformations that are compliant to the Meta Object Facility (MOF) standard [31]. The Kermeta workbench allows developers to specify well-formedness rules (i.e., invariants) on models. These rules can be

expressed using an OCL-like specification language. The Kermeta workbench provides several APIs for evaluating OCL-like invariants against instances of a class model. It returns warning information to indicate the invariant that is violated by the given instance.

3.2 Analyzing Operation Contracts of UML Class Models

In previous work [44] we developed a class model analysis approach that uses the Alloy Analyzer [18] to find scenarios (sequences of operation invocations) that start in valid states (states that satisfy the invariants in the class model) and end in invalid states (states that satisfy the negation of the invariants). The analysis uses the operation contracts to determine the effects operations have on the state. If analysis uncovers a sequence of operation calls that moves the system from a valid state to an invalid state, then the designer uses the trace information provided by the analysis to determine how the operation contracts should be changed to avoid this scenario.

The approach uses UML-to-Alloy and Alloy-to-UML transformations to shield the designer from the back-end use of the Alloy language and analyzer. The transformations used in the approach build upon the UML2Alloy transformation tool [8][2][1] developed at the University of Birmingham. The work proposed in [44] 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 [44] also builds upon our previous work on the Scenario-based UML Design Analysis (ScUDA) approach [51][49][50][48]. 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 [44] 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 [44] 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?”.

Chapter 4

The Proposed Slicing Techniques

In this chapter we propose class model slicing techniques to decomposing large class models into model fragments where each model fragment contains just those elements needed to perform the analysis. The slicing techniques can be used to improve the scalability of two class model analysis techniques, (1) one that involves checking the consistency between an object configuration and a class model with specified invariants (Section 4.2) and (2) the other that involves analyzing sequences of operation invocations to uncover invariant violations (Section 4.3).

In the reminder of this chapter we present an example that will be used to illustrate the class model slicing techniques (Section4.1) and describe the slicing techniques (Section 4.2 and Section 4.3).

4.1 Illustrating Example

We will use the Location-aware Role-Based Access Control (LRBAC) model, proposed by Ray et al. [32] [33] [34], to illustrate the model slicing technique. LRBAC is an extension of Role-Based Access Control (RBAC) [36] 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

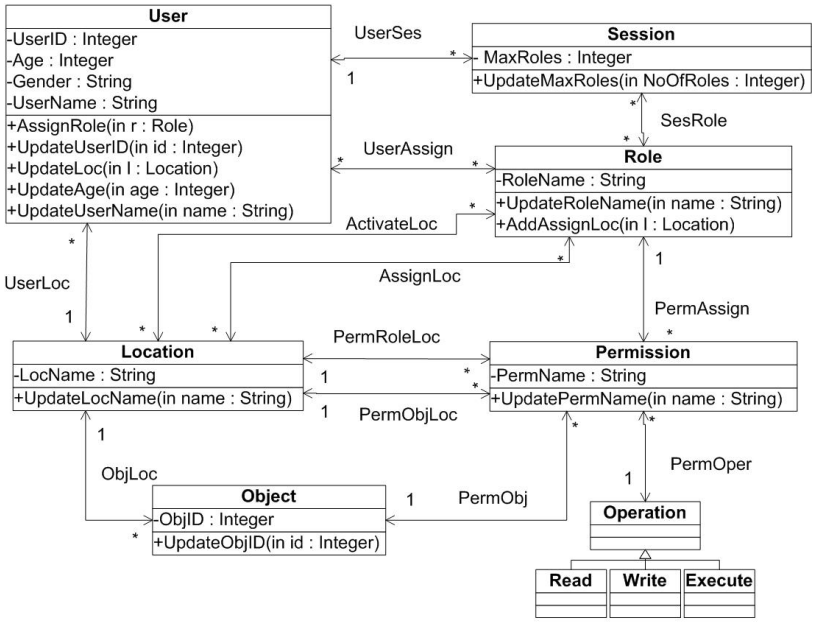


Figure 4.1: A Partial LRBAC Class Model

that is associated with locations can be assigned to a user only if the user is in a location in which the role can be assigned. A user can create a session and activate his assigned roles in the session. A role can be activated in a session only if the user that creates the session is in a location in which the role can be activated. Figure 4.1 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.

Operation contracts and invariants in the LRBAC model are specified using the OCL. For example, the OCL contracts for operations *UpdateUserID*, *AssignRole*, *UpdateLoc*, and *UpdateAge* are given below:

// Update a user's ID

Context *User::UpdateUserID(id:Int)*

// Precondition: the user's id is not equal to the input id

Pre: *self.UserID != id*

// Postcondition: the user's id is equal to the input id

Post: *self.UserID = id*

// 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)*

// 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)*

// Update a user's age

Context *User::UpdateAge(age:Int)*

// Precondition: the input age must be greater than 0

Pre: *age > 0*

// Postcondition: the user's age is equal to the input age

Post: *self.Age = age*

Examples of OCL invariants for the LRBAC model are given below:

// Each user has a unique ID.

Context *User* ***inv UniqueUserID::***

User.allInstances()→forAll(u1, u2:User|u1.UserID = u2.UserID implies u1 = u2)

// 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))

// 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()

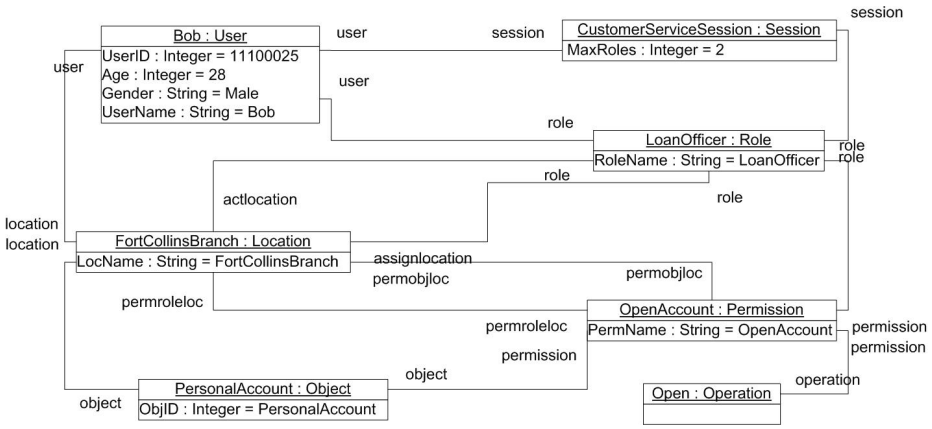


Figure 4.2: An Instance of LRBAC Class Model

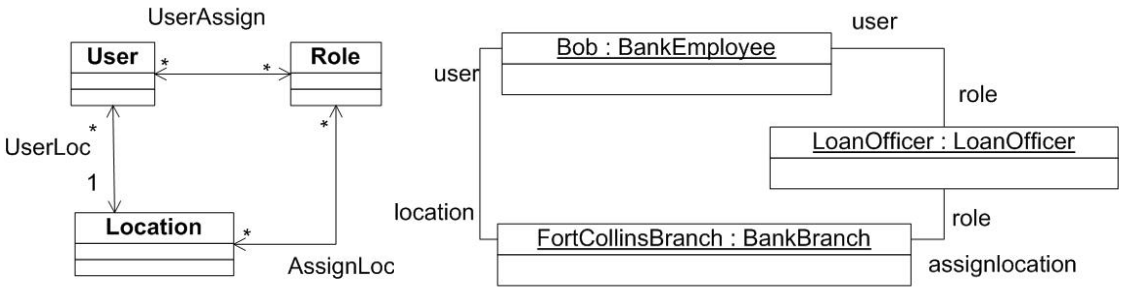
4.2 Co-slicing Class Model and Object Configuration

The slicing technique described in this section is used to extract a fragment from a class model and an object configuration that conforms to the class model, where the fragment contains class model elements and object configuration elements needed to analyze an invariant defined on the class model.

The input to the technique includes a class model with an invariant expressed in the OCL, and an object configuration that conforms to the class model. The technique has two major steps. In the first step, the invariant is used as a slicing criterion to extract a fragment from the input class model. The class model fragment consists of the elements that are only referenced by the invariant. In the second step of the technique, the class model fragment is used to generate a fragment from the input object configuration. The objects, links and slots that are not instances of elements involved in the class model fragment are removed from the input object configuration.

4.2.1 Slicing a Class Model w.r.t. an Invariant

Figure 4.2 shows an instance of the LRBAC class model givin in Figure 4.1. Suppose that we plan to check if the object configuration in Figure 4.2 satisfies the *CorrectRole-*



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

Figure 4.3: Sliced LRBAC Model and Object Configuration

Assignment invariant defined on the class model in Figure 4.1. The *CorrectRoleAssignment* invariant is used as a slicing criterion by the technique described in the section. The part of a class model that is referenced by the slicing criterion is needed to be identified. This can be done by performing a dependency analysis between the invariant and the class model.

Dependencies between an OCL invariant and class model elements are computed by traversing the syntax tree of the invariant. For example, consider the analysis of the *CorrectRoleAssignment* invariant (the invariant is given in Section 4.1). The expression *self*.*UserAssign* is an association end call expression and it returns a set of roles assigned to the user (referred to by *self*). There is thus a dependency between this invariant 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 invariant references and thus depends on, the following classes: *User*, *Role* and *Location*. A similar analysis can done for attributes, operations and other types of class model elements. Figure 4.3a shows a fragment of the LRBAC model that is referenced by the *CorrectRoleAssignment* invariant.

Algorithm 1: Extract an Object Configuration Fragment

```
1:  Input: An object configuration, OC, and a sliced class model, SCM
2:  Output: An object configuration fragment that conforms to the sliced class model
3:  Algorithm Steps:
4:  Set MElmts = {};
5:  for each model element ME in SCM do
6:     MElmts = MElmts ∪ ME;
7:  end for
8:  for each element E in OC do
9:     if E's metaclass not in MElmts then
10:       Remove E from OC;
11:    end if
12: end for
13: Return OC;
```

4.2.2 Slicing an Object Configuration

Figure 4.3b shows an object configuration fragment that conforms to the sliced LRBAC class model given in Figure 4.3a. The object configuration fragment is generated from the object configuration given in Figure 4.2 using Algorithm 1.

The algorithm first computes *MElmts* (see lines 4-7), a set of class model elements, where each member is an element of the input class model *SCM*. The algorithm then traverses each element of *OC* (see lines 8-12). For an element of *OC*, if its metaclass is not a member of *MElmts*, the algorithm removes the element from *OC*. The output of the algorithm is an object configuration fragment that conforms to the sliced class model.

4.3 Slicing a Class Model with OCL Constraints

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 [44], we developed a class model analysis technique that uses the Alloy Analyzer [18] to find scenarios (sequences of operation

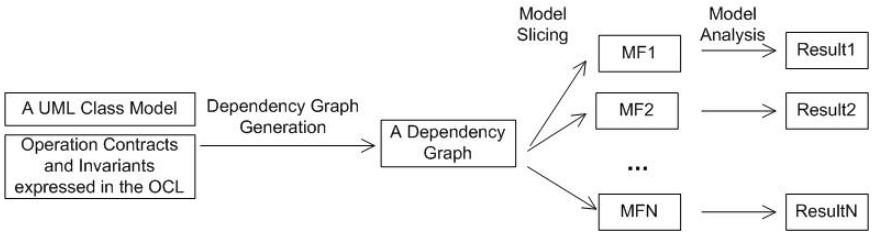


Figure 4.4: Technique Overview

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

The model slicing technique described in this section is used to decompose a large class model into fragments, where each fragment contains model elements needed to analyze a subset of the invariants and operation contracts in the class model. Fig. 4.4 shows an overview of the slicing technique.

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

In the second step of the technique, the dependency graph is used to generate slicing

Table 4.1: 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

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 describe the process for generating a dependency graph and the slicing algorithm used to decompose the class model into model fragments.

4.3.1 Constructing a Dependency Graph

Dependencies among invariants, operation contracts and model elements are computed by traversing the syntax tree of the OCL invariants and operation contracts. For example, consider the analysis of the operation contract for *AssignRole* (the contract is given in Section 2). The expression *self*.*UserAssign* is an association end call expression and it returns a set of roles assigned to the user (referred to by *self*). There is thus a dependency between this contract and the class *Role*. The expression *self*.*UserLoc* returns a user's

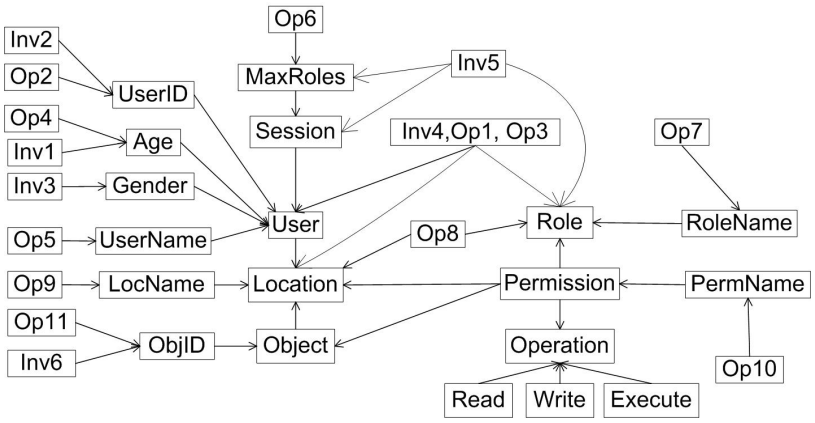


Figure 4.5: A Dependency Graph

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 4.1 lists the referenced classes and attributes for the contracts and invariants defined in the LRBAC model.

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

Figure 4.5 shows a graph that describes the dependency relationship among classes, attributes, operations and invariants of the LRBAC class model described in Fig. 4.1.

Algorithm 2 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 4.1 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. 4.5. Invariant $Inv5$ in Table 4.1 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. 4.5 since invariant $Inv5$ also references class *Role* in its specification through the navigation from class *Session* to class *Role*.

4.3.2 Analyzing a Dependency Graph

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

Algorithm 2: Dependency Graph Generation Algorithm

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

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 3, 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 [38]. Model elements that are involved in a *local* analysis problem are referred to as *local analysis model elements*. For example, operation *UpdateUserID* in Fig. 4.1 is used to modify the value of attribute *UserID* in class *User*, and invariant *Inv2* defines the uniqueness constraint on *UserID*. The invocation of operation *UpdateUserID* may or may not violate the constraint specified in *Inv2*, but it will not violate other invariants because *UserID* is not referenced by other operations or invariants. Thus

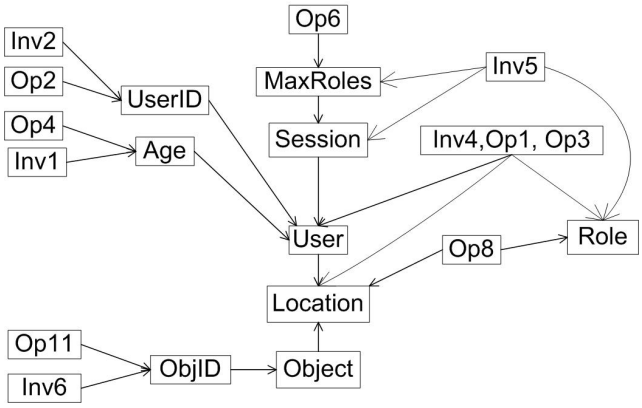


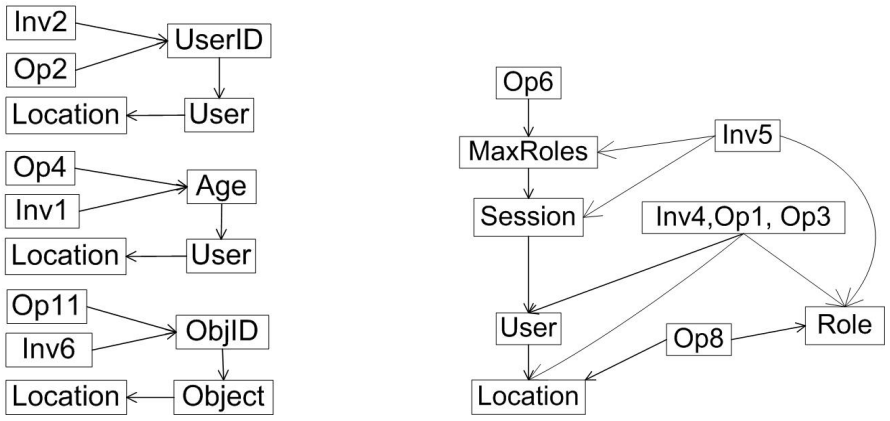
Figure 4.6: A Dependency Graph Representing a LRBAC Model with the *Irrelevant Model Elements* Removed

an analysis that involves checking if an invocation of *UpdateUserID* violates *Inv*2 can be performed within the boundary of *User*. Model elements that are involved in *local* analysis problems are identified using the process described in Algorithm 4. 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 5.

4.3.2.1 Identifying *Irrelevant Model Elements*:

Algorithm 3 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*.



(a) Local Analysis Model Elements identified by Algorithm 4

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

Figure 4.7: Dependency Graphs Representing Model Fragments Extracted from the LRBAC Model in Fig. 4.1

The vertices not in *ARVSet* represent the *irrelevant model elements* that need to be removed from the class model.

Figure 4.6 shows a dependency graph representing a LRBAC model with the analysis irrelevant model elements removed. Lines 6-14 of Algorithm 3 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*.

4.3.2.2 Identifying *Local Analysis Model Elements*:

Algorithm 4 is used to identify fragments representing local analysis problems. The algorithm first computes a set of attribute and class vertices, *LocalVSet*, that are involved

Algorithm 3: Irrelevant Model Elements Identification Algorithm

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

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.

The dependency graph in Fig. 4.6 is decomposed into several subgraphs, as shown in Fig. 4.7a and Fig. 4.7b, using Algorithm 4. Each dependency graph in Fig. 4.7a

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. 4.6. The vertices that directly depend on vertex *ObjID* are $Op11$ and $Inv6$, and they are moved from *DG* to a new dependency graph. Vertex *ObjID*'s DFS tree consists of *ObjID*, *Object* and *Location*, and they are copied from *DG* to the new dependency graph. *ObjID* is then removed from *DG*. Note that vertex *Object* becomes analysis-irrelevant in *DG* after vertex *ObjID*, $Op11$, and $Inv6$ have been removed from *DG*. Thus it is necessary to perform Algorithm 3 on *DG* to remove the analysis-irrelevant vertices, as indicated by Line 25.

4.3.2.3 Decomposing the Dependency Graph:

Algorithm 5 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 5 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 3) 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. 4.7b is {*MaxRoles, User, Location, Role*}. Thus *Col* for the graph is $\{\{Op6, In5\}, \{In4, Op1, Op3\}, \{In4, Op1, Op3, Op8\}, \{In4, Op1, Op3, Op8, In5\}\}$.

Line 6 uses the union-find algorithm described in [11] 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. 4.7b becomes $\{Op6, In5, In4, Op1, Op3, Op8, In5\}$ with the union-find algorithm being used.

Algorithm 4: Local Analysis Problem Identification Algorithm

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

Lines 7-16 use each disjoint set, *S*, in *Col* to construct a new dependency graph from the input dependency graph *DG*. Lines 8-13 build a forest for *S* from each DFS tree of a vertex in *S*. Lines 14-15 create a new dependency graph that consists of all vertices in the forest. Since *Col* for the graph shown in Fig. 4.7b has only one disjoint set, the forest generated from the disjoint set consists of all the vertices in the dependency graph, indicating that the graph in Fig. 4.7b is the minimum dependency graph that cannot be decomposed further.

Algorithm 5: Dependency Graph Decomposition Algorithm

1: Input: A dependency graph, *DG*, produced from the original graph by Algorithm 4
2: Output: A set of dependency graphs
3: Algorithm Steps:
4: Recompute *ARClsAttrVSet* for *DG* using Algorithm 3;
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 [11] 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*;

Chapter 5

Preliminary Evaluation

The objective of this evaluation is to analyze the effectiveness of the slicing techniques described in the proposal. Specifically the preliminary evaluation aims to investigate the following questions:

1. Can our slicing technique improve the efficiency of a class model analysis approach that involves checking the consistency between an object configuration and a class model with specified invariants (Section 5.1)?

2. Can our slicing technique improve the efficiency of a class model analysis approach that involves analyzing sequences of operation invocations to uncover invariant violations (Section 5.2) ?

5.1 Evaluating Co-slicing of Class Model and Object Configuration

The slicing technique described in Section 4.2 takes as input a class model with an invariant and an object configuration conforming to the class model, and produce a fragment that contains part of the class model and object configuration that are referenced by the invariant. It can be used to speed up the consistency checking process for large class models and object configurations. In the remainder of this section we describe a

criterion used to evaluate the effectiveness of the slicing technique (Section 5.1.1) and present the results of a preliminary evaluation we performed on the slicing technique (Section 5.1.2 and Section 5.1.3).

5.1.1 Evaluation Criterion

The effectiveness of the slicing technique is determined by (1) the slicing ratio (i.e., the proportion of the elements that are removed from the original model) and (2) the speedup achieved for the analysis time. The slicing ratio on class model and object configuration can be calculated using the equations below:

$$\textit{Model Slicing Ratio (MSR)} = \frac{\textit{RemovedME}}{\textit{TotalME}},$$

$$\textit{Object Configuration Slicing Ratio (OCSR)} = \frac{\textit{RemovedOCE}}{\textit{TotalOCE}},$$

where *RemovedME* refers to the number of model elements that are removed from a class model by the slicing technique, *TotalME* refers to total number of model elements in a class model, *RemovedOCE* refers to the number of objects, links and slots that are removed from an object configuration by the sling technique, and *TotalOCE* refers to total number of objects, links and slots in an object configuration.

The analysis time speedup is calculated using the equation below:

$$\textit{Analysis Time Speedup (ATS)} = 1 - \frac{\textit{SAT}}{\textit{AT}},$$

where *SAT* refers to the analysis time used for the sliced class model/object configuration, and *AT* refers to the analysis time used for the unsliced model/object configuration. Give a class model with an invariant and an object configuration that conforms to the class model, the slicing technique described in Section 4.2 is effective

if more class model/object configuration elements can be removed in the slicing process (i.e., high slicing ratio) and less time is required for analyzing the sliced class model/object configuration (i.e., high speedup).

5.1.2 Analysis Results of Unsliced Class Models and Object Configurations

The class model/object configuration consistency checkers used in the evaluation include the USE tool, the Alloy Analyzer, and the Kermeta workbench (see the related background material given in 3.1). The class models used for the evaluation were selected from both academia (e.g., Location-aware Role-Based Access Control Model [33]) and industry (e.g., OCL metamodel). The selected class models conform to UML standard and are expressed in the EMF Ecore form. The invariants defined on class models are specified using the OCL. A list of class models used for the evaluation is given below:

1. OCL metamodel [42] from the Eclipse OCL project;
2. Alloy metamodel [18] from the Software Design Group at MIT;
3. LRBAC class model [33] from the Database Security Group at CSU;
4. ScUDA metamodel [49] from the Software Engineering Group at CSU.

Note that a metamodel is a particular type of a class model that describe the structure of a collection of class models.

The Kermeta workbench can directly take as input Ecore models while the USE tool and the Alloy Analyzer cannot. We have thus developed an Ecore-to-USE transformer that translates an Ecore class model into the USE specification, and an Ecore-to-Alloy transformer that translates an Ecore class model into the Alloy specification. Both transformers are developed using the Kermeta workbench.

Table 5.1: Class Model Size

Model	Class	Enum	DataType	Attribute	Reference	Operation	Total
OCL	51	5	1	25	58	0	140
Alloy	41	6	3	23	36	0	109
LRBAC	7	0	0	5	24	14	50
ScUDA	9	0	0	2	15	0	26

Table 5.2: Object Configuration Size

Object Configuration	Object	Link	Slot	Total
OCLOC	840	800	8040	9680
AlloyOC	80	80	24	184
LRBACOC	3213	4430	0	7643
ScUDAOC	105	100	5	210

Table 5.1 shows the size of class models used for the evaluation. The size of a class model is calculated in terms of total number of elements (e.g., classes, attributes, references). Note that Ecore uses references to support the association and composition concepts defined in the UML.

The object configurations used for the evaluation are created using the Kermeta workbench. For example, given a class model, we first load it into the Kermeta workbench, and then write a Kermeta program to generate an object configuration from the class model. We have manually translated Kermeta programs used to generate object configurations into USE commands and Alloy predicates.

Invariants defined on class models are specified using the OCL. Since both the Kermeta workbench and the Alloy Analyzer cannot take as input OCL invariants, we have manually translated OCL invariants into the Kermeta program and Alloy facts.

Table 5.2 shows the size of object configurations used for the evaluation. The size of an object configuration is calculated in terms of total number of objects, links and slots.

Table 5.3: Analysis Results of Unsliced Class Models and Object Configurations

Class Model/Object Configuration	Time(ms) in USE	Time(ms) in Kermeta	Time(ms) in Alloy
OCL/OCLOC	7	3006	Translation capacity exceeded
Alloy/AlloyOC	9	1065	Cannot get result within three hours
LRBAC/LRBACOC	6	450	1594928
ScUDA/ScUDAOC	5	191	636097

Table 5.3 shows the results of an analysis we performed on a laptop computer with 2.17 GHz Intel Dual Core CPU, 3 GB RAM and Windows 7. In the analysis each tool (i.e., USE, Kermeta, Alloy) is used to check if an object configuration satisfies an invariant defined on a class model to which the object configuration conforms. The time used for consistency check is measured in terms of milliseconds.

The analysis results showed that the USE tool has fairly good performance on both large class models and object configurations, and the execution time of the Alloy Analyzer could be exponential on the size of the model. The Alloy Analyzer cannot deal with neither large models nor large object configurations. Even for a small class model, ScUDA, it would take more than 10 minutes for the Alloy Analyzer to perform the analysis. Note that the error "translation capacity exceeded" means a class model encoded in the Alloy specification cannot be translated into the SAT formula because the total elements of OCL/OCLOC exceeds the maximum number of elements allowed for the Alloy Analyzer.

5.1.3 Analysis Results of Sliced Class Models and Object Configurations

We applied the slicing technique to the class models and object configurations given in Table 5.1 and Table 5.2. Table 5.4 shows the size of sliced class models, and Table 5.5 shows the size of sliced object configurations. The slicing results showed the slicing

Table 5.4: Sliced Class Model Size

Sliced Model	Class	Enum	DataType	Attribute	Reference	Operation	Total	MSR (%)
OCL	1	0	0	1	0	0	2	98.57
Alloy	2	0	1	0	1	0	4	96.33
LRBAC	2	0	0	0	2	0	4	92.00
ScUDA	2	0	0	1	0	0	3	88.46

Table 5.5: Sliced Object Configuration Size

Sliced Object Configuration	Object	Link	Slot	Total	OCSR (%)
OCLOC	20	0	20	40	99.59
AlloyOC	80	72	8	160	13.04
LRBACOC	110	100	0	210	97.25
ScUDAOC	30	25	5	60	71.43

technique can significantly reduce the size of class models and object configurations being analyzed. Note that the slicing ratio is high for all the class models and most of the object configurations.

We used the consistency checkers to analyze the sliced class models and object configurations. Table 5.6 shows the time used for analyzing sliced class models and object configurations. Note that the speedup for OCL/OCLOC cannot be calculated since unsliced OCL/OCLOC cannot be translated into SAT formula and the analysis time used for unsliced OCL/OCLOC cannot be determined. The speedup for Alloy/AlloyOC is approximately 100 % because the analysis time used for unsliced Alloy/AlloyOC is significantly larger than the time used for sliced Alloy/AlloyOC.

The evaluation results in Table 5.6 showed that the slicing technique can significantly improve the scalability of the Alloy Analyzer. For example, the analysis time speedup for sliced Alloy/AlloyOC, LRBAC/LRBACOC and ScUDA/ScUDAOC is 100%, 99.48%, 99.93% respectively, and it only took 42 milliseconds for analyzing sliced OCL/OCLOC. The slicing technique also works well for the Kermeta work-

Table 5.6: Analysis Results of Sliced Models and Object Configurations

Sliced Class Model / Object Configuration	Time(ms) in USE	ATS for USE (%)	Time(ms) in Kermeta	ATS for Kermeta (%)	Time(ms) in Alloy	ATS for Alloy (%)
OCL/OCLOC	7	0	448	85.10	42	?
Alloy/AlloyOC	5	44.44	193	81.88	6130	≈ 100
LRBAC/LRBACOC	5	16.67	252	44.00	8296	99.48
ScUDA/ScUDAOC	5	0	123	35.60	443	99.93

bench. The analysis time speedup for sliced model and object configurations varies from 35.60% to 85.10% in the Kermeta workbench. The slicing technique may or may not improve the scalability of the USE tool, depending on the input class model and object configuration. For example, the analysis time achieves 44.44% speedup on Alloy/AlloyOC and 16.67% speedup on LRBAC/LRBACOC in the USE tool, while the analysi time achieves 0 speedup on both OCL/OCLOC and ScUDA/ScUDAOC.

The evaluation also showed that the analysis results are preserved by the slicing technique. The analysis results of the sliced models and object configurations are the same as the unsliced ones. For example, all the consistency analysis tools confirmed that both the unsliced LRBAC object configuration and the sliced LRBAC object configuration satisfy the *CorrectRoleAssignment* invariant (see Section 4.1) defined on the LRBAC class model.

5.2 Evaluating Contract-aware Slicing of Class Model

We developed a research prototype to investigate the feasibility of developing tool support for the slicing technique. The prototype was developed using Kermeta [21], an aspect-oriented metamodeling tool. The inputs to the prototype are (1) an EMF Ecore [43] file that describes a UML design class model, and (2) a textual file that contains the OCL invariants and operation specifications. The prototype produces a list of model

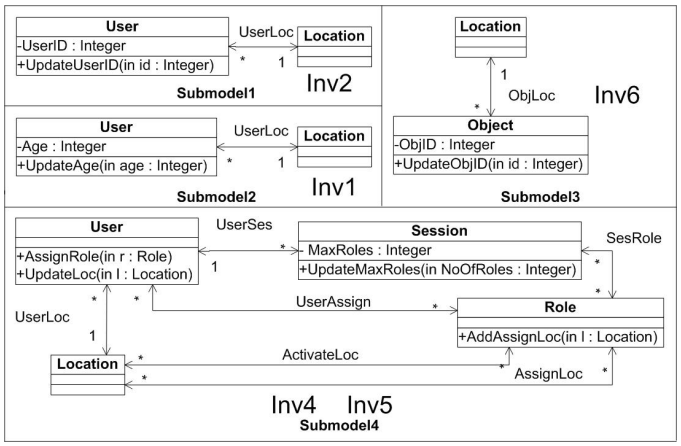
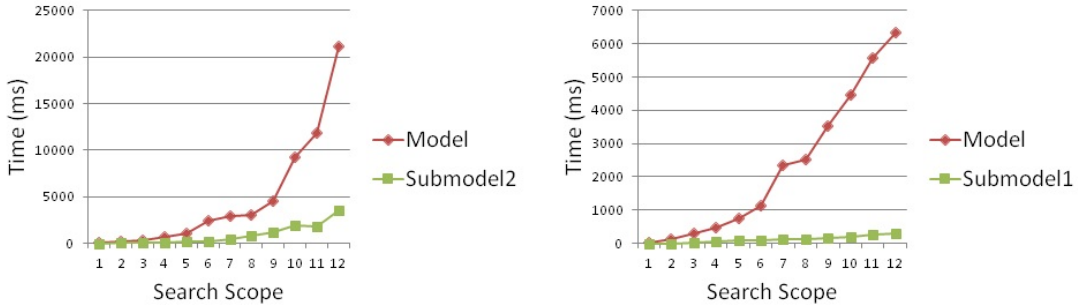


Figure 5.1: A List of Model Fragments Generated from the LRBAC Model in Fig. 4.1

fragments extracted from the input model. The prototype implementation uses a visitor pattern based transformation approach to generate a dependency graph from a UML class model with OCL invariants and operation specifications. Two Kermeta visitor classes, CD2DG (i.e., class model to dependency graph visitor) and OCL2DG (i.e., OCL invariants/operation specifications to dependency graph visitor) are defined in the prototype. Each visitor is used to traverse a source model (e.g., a class model or an OCL specification) based on its abstract syntax tree (e.g., an Ecore metamodel or an OCL metamodel) and to directly transform each element of the source model to a corresponding vertex of the target dependency graph.

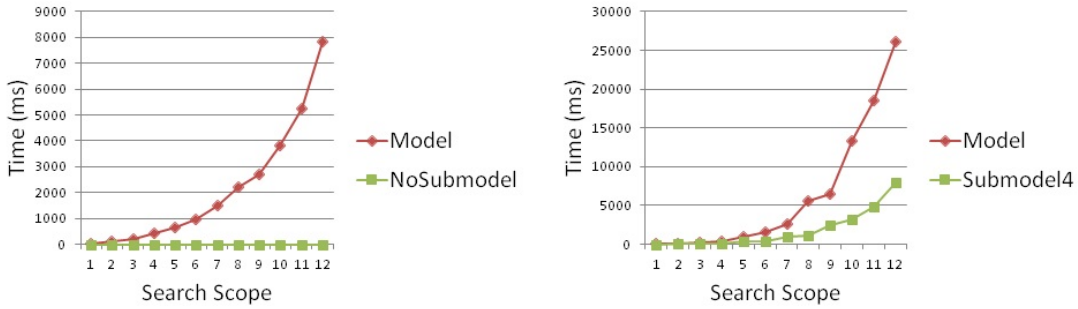
Figure 5.1 shows four model fragments extracted from the LRBAC model in Fig. 4.1. Each model fragment corresponds to a dependency graph in Fig. 4.7. The model fragments and the unsliced model were analyzed against the invariants defined in Table 4.1 by the Alloy Analyzer (version 4.2 with SAT4J), on a laptop computer with 2.17 GHz Intel Dual Core CPU, 3 GB RAM and Windows 7.

Figure 5.2 shows the results of a preliminary evaluation we performed on the unsliced model and the model fragments. Each subfigure in 5.2 has an x axis, namely *SearchScope*, indicating the maximum number of instances the Alloy Analyzer can pro-



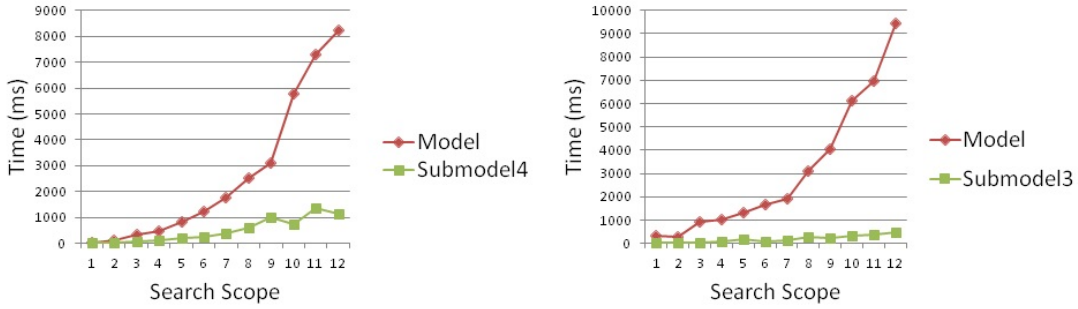
(a) Analyzing Model Fragment against Inv1

(b) Analyzing Model Fragment against Inv2



(c) Analyzing Model Fragment against Inv3

(d) Analyzing Model Fragment against Inv4



(e) Analyzing Model Fragment against Inv5

(f) Analyzing Model Fragment against Inv6

Figure 5.2: Analyzing Unsliced Model and Model Fragments against each Invariant of Table 4.1

duce for a class, and a *y* axis, namely *Time*, showing the total analysis time (in millisecond) for building the SAT formula and finding an Alloy instance.

For example, Fig. 5.2a shows the time used to analyze the unsliced model *Model* and the model fragment *Submodel*2 against the invariant *Inv*1. The difference between the

time used for analyzing *Model* and that for *Submodel*2 is relatively small when the Alloy search scope is below 5. For a search scope above 10, the time used for analyzing *Model* becomes significantly large while that for *Submodel*2 is still below 5000 ms. Fig. 5.2c shows that the invariant *Inv*3 was not analyzed in any model fragment (the analysis time remains at 0). This is because *Inv*3 was removed after the dependency analysis identified the invariant as an analysis-irrelevant element.

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

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

Chapter 6

Discussion and Future Work

In this chapter we discuss limitations of the slicing techniques and the preliminary evaluation (Section 6.1 and Section 6.2). We also summarize the future work to address these limitations (Section 6.3).

6.1 Limitation of the Slicing Technique

The slicing technique on class model and object configuration is limited in its ability to produce a smaller fragment from a larger class model and object configuration w.r.t. the slicing criterion (i.e., an invariant defined the class model). There may be invariants that reference all elements of a class model. In this case the slicing technique cannot ensure that a smaller fragment will be produced from a class model and object configuration. An empirical study by Juan et al. [10] showed that in practice most of invariants defined on a class model only reference a subset elements of the model. Thus our slicing technique should work for many class models used in practice.

The slicing technique on class model with invariants and operation contracts is limited in its ability to produce smaller model fragments from a large class model w.r.t. the OCL constraints (i.e., invariants and operation contracts). There may be constraints that reference all model elements of a class model and thus require the entire class model to be present when analyzed (reflecting a very tight coupling across all model elements).

In this case, the slicing technique described in this proposal does not ensure that more than one independently analyzable fragments will be produced from a class model.

6.2 Limitation of the Preliminary Evaluation

One limitation of the preliminary evaluation is that the evaluation results may not be generalized since we used only a few models and object configurations to evaluate the slicing technique. For example, we used only four class models, four object configurations and four invariants in the evaluation described in Section 5.1, and only one class model that includes 11 operations and six invariants in the evaluation described in Section 5.2. Currently only one object configuration is manually generated for each class model in the evaluation.

6.3 Future Work

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

Another objective of our future work is to address the limitation of the preliminary evaluation. We plan to apply the slicing technique to more (1) class models and object configurations and (2) class models with OCL constraints.

We are currently using class models (e.g., UML metamodels) that have a substantial number of invariants and operation contracts to evaluate the slicing technique. These class models were selected from state-of-the-art model repositories (e.g., ReMoDD[46] and Metamodel Zoos[45]). We also selected class models developed by the students from the software engineering courses at CSU. We are currently using the approach described in [9] to automatically generate object configurations that conform to a given

class model.

We also plan to use a variety of OCL constraints as slicing criteria in the evaluation. Specifically we are currently investigating how we can use OCL benchmark given in [17] to generate complex OCL constraints.

Chapter 7

Research Plan

Figure 7.1 shows a planned timeline for the proposed research work. Task 1 refers to a state-of-the-art survey on model slicing techniques. Task 2 refers to a rigorous technique that supports co-slicing of UML class models and object configurations. Task 3 refers to a rigorous technique that supports slicing of UML class models including OCL constraints. Task 4 refers to a platform that provides implementations of two proposed class model slicing techniques. Task 5 refers to an evaluation and validation framework for the proposed slicing techniques. Task 6 refers to the additional work described in Section 6.3 to complete the dissertation.

By the time of the proposal submission, we have accomplished tasks 1, 2 and 3. Task 4 is still in progress, and will be finished by the spring of 2014. We will start to work on task 5 and 6 in the end of 2013. Finally, a PhD dissertation will be produced in 2014.

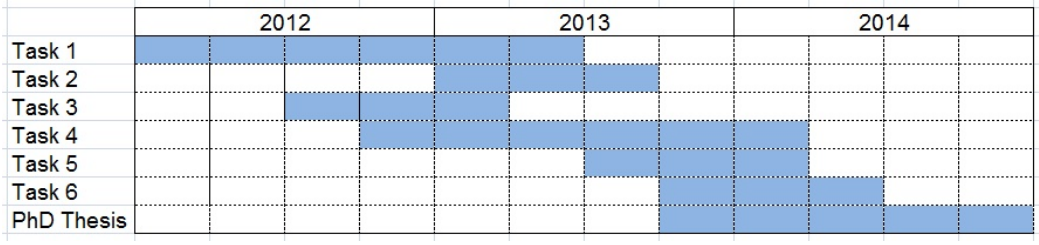


Figure 7.1: A Planned Timeline for the Research Work

REFERENCES

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

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

[3] K. Androutsopoulos, D. Binkley, D. Clark, N. Gold, M. Harman, K. Lano, and Z. Li. Model projection: simplifying models in response to restricting the environment. In *Proceedings of 33rd International Conference on Software Engineering (ICSE)*, pages 291–300. IEEE, 2011.

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

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

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

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

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

[9] J. Cadavid, B. Baudry, and H. Sahraoui. Searching the boundaries of a modeling space to test metamodels. In *Proceedings of the IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pages 131–140. IEEE, 2012.

[10] 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.

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

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

[13] 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.

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

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

[16] 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.

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

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

[30] 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.

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

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

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

[34] 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.

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

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

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

[38] 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.

[39] 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.

[40] 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.

[41] 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.

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

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

[44] 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.

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

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

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

[48] 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.

[49] 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.

[50] 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.

[51] 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.

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