Context-Free Slicing of UML Class Models

Huzefa Kagdi, Jonathan I. Maletic, Andrew Sutton
Department of Computer Science
Kent State University
Kent Ohio 44242
{hkagdi, jmaletic, asutton}@cs.kent.edu

Abstract

The concept of model slicing is introduced as a means to support maintenance through the understanding, querying, and analysis of large UML models. The specific models being examined are class models as defined in the Unified Modeling Language (UML). Model slicing is analogous to classical program slicing. In program slicing the computational behavior at a statement involving variables specified by the slicing criteria defines the context. Since UML class models do not explicitly embody any behavioral aspect by themselves, models slices are computed in a context-free manner. The paper defines context-free model slicing and presents an algorithm for computing slices on class models. Concrete applications of model slicing to address particular software maintenance questions are presented to support the usefulness and validity of the method.

1. Introduction

There is an ever increasing importance being put on design models to support the evolution of large software systems. Design models are being maintained and updated from initial development as well as being reverse engineered to more accurately reflect the state of evolving systems.

The advent of usable tools and standards to support design models is obviously making the storage and maintenance of these models realizable for development teams. The Unified Modeling Language (UML) supplies us with a number of models that are commonly used in the construction and long-term evolution of large-scale software systems. Tools such as Rational Rose, ArgoUML, etc. greatly assist in the representation, development, and maintenance of these models. In particular, UML class models are very useful for understanding and maintaining existing systems, mostly due to their concrete mapping to the source-code. Unfortunately, UML tools do not support source-to-model consistency checking very well, and they often fail to accurately reverse engineer class models. However, project teams have integrated these tools into their software processes and a great deal of effort is being expended to reverse engineer these design models and to maintain consistency of these large models with the actual source.

Herein lies the problem. A class model for a large system is typically comprised of thousands of classes and relationships. Viewing the entire model at one time is impractical and typically of little use for a particular maintenance task.

In reality, designs are initially developed and presented via a large-number of small UML class diagram (10-15 classes). Each diagram (sub-model) typically describes a feature or concern of the system design or functionality. The union of these diagrams represents the entire system design.

In the case of a purely reverse engineered model no smaller diagrams exist. Meaningful diagrams are constructed by hand from the model as needed. Alternatively, if diagrams do exist from initial development they may be wholly inadequate for a given maintenance task.

It is now becoming apparent that there is need for methods and tools to assist in generating sub-models based on specific maintenance tasks. This is much akin to the tools we currently have for examining source code. There are many mature tools for querying and browsing source code. Such tools include fact-extractors, program dependency graphs generators, call graph analyzers, and program slicers. These tools are used to help reduce the amount of code that needs to be examined and understood to address the problem at hand. Our work is aimed at providing similar support for maintenance at the design level (i.e., for UML class models).

There are few, if any, existing methods (or tools) for supporting the automated or semi-automated extraction of meaningful subsets of a class model. Currently, this is done manually. An engineer must wade through the entire class model and (using some tool) construct a specific class diagram within the context of this model. While a diagram may exist that is close to what is desired (e.g., the entire class hierarchy for a particular concept) it may be far too unwieldy and include many classes of little interest or consequence to the problem at hand (e.g., entity classes contained by all classes in an inheritance hierarchy).

In the work presented here, we introduce a method to automatically generate a subset of a UML class model based on a user-defined criterion. The goal of this work is to allow us to automatically extract a pertinent and meaningful UML class diagram from a very large UML class model. In the next section (2) we describe and formally define our approach to UML class model slicing.

Concrete applications of this approach are given in section 3. In this section we formulate a number of specific maintenance questions in terms of model slicing. The questions deal with such activities as:

- discovering relationships between classes;
- finding classes that implement a method with a known bug; and
- finding classes augmented by metric values to identify potential refactorings.

Related work on program slicing, examining subsets of UML diagrams, and feature location are discussed in section 4. Conclusions are given in section 5.

2. UML Model Slicing

Our method is rooted in the classical definition of program slicing but extends that concept to address the multi-graph nature of the UML metamodel. In general, we term this approach *model slicing*. However, since class models are devoid of explicit behavioral information (by themselves) we further define the concept of context-free model slicing. Program slicing has the implicit context of the definition-use relationship with respect to a supplied slicing criterion. In model slicing of a UML class model, we must specify some sort of non-behavioral aspect to construct the slice. In short, model slices are defined via a generalized slicing criterion that is specified with predicates over the model's features.

Program slicing, as defined in [20], takes a program and a slicing criteria to compute a slice or subset of the source code. More formally, given a slicing criterion, $s(v, n)$ with a set of variables v and a location of a statement of interest s, program slicing determines a set of statements contributing directly or indirectly to the values of variables, v, before the statement s is executed. Those resultant statements comprise the program slice. Statements in a program slice have a specific behavioral context. In this case, the behavior is the statements affected by or affecting the states of the variables involved in a computation at a particular point in a program. These statements included in the program slice are those extracted from the investigation of the definition-use relationship (i.e., statements with definition and usage of variables given in the slicing criteria).

It should be noted the definition-use relationship is the only relationship of interest in program slicing. This is evident in the definition of the program-slicing criteria. The slicing criterion does not provide any means for the explicit specification of relationships other than definition-use relationship. The usage of this relationship is implicitly assumed. This assumption restricts program slicing to the singular relationship among the elements (statements) of the source code.

UML model slicing extends this concept of program slicing via a generalized, albeit more complex, slicing criteria. These extensions elevate the capabilities of program slicing from the source-code level to UML structural models (i.e., class models) and behavioral models (i.e., sequence, collaboration, and state behavior models). All elements and relationships defined in the UML metamodel can be used in the computations of a model slice. This includes elements such as classes, packages, components and operations, and relationships such as associations, dependencies and generalizations. The slicing criteria for the extended domain must account for all the elements and relationships available in the UML metamodel.

Unlike program slicing, model slicing does not necessarily require the physical location of an element of interest (i.e., an observation point of a behavior). UML class models, representing only abstracted structural elements, contain no behavioral elements (e.g., instances of classes or statements). However, other views such as sequence diagrams, object diagrams, and collaborations define contexts in which objects may be explicitly located. In these cases, we define *context* to be the location of the object. The context can be a particular set of scenarios in which a set of objects are involved or a particular range in the lifeline of a set of objects when dealing with an interaction model such as one pictured by a sequence diagram.

Here, we distinguish between slicing of models that require or do not require context information and introduce the terms *context-free* and *context-sensitive* slices. The context-free slices are applicable to models that do not require a context for the computation of a model slice. Context-sensitive slices are applicable to models that do require such context information. Here we focus on the definition of context-free model slicing and reserve the definition of context-sensitive slicing for future work (as it requires the former definition at the very least).

2.1. Context-free Model Slice

A context-free model slice is defined primarily to encapsulate static and structural aspects of a UML model and precludes the inclusion of behavioral, computation, or interaction information. For the purpose of model slicing, we define a model, *M*, as a directed multi-graph $M = (E, R, \Gamma)$ where

- $E = \{e_1, e_2,.., e_n\}$ is the finite set of elements,
- $R = \{r_1, r_2,...,r_m\}$ is the finite set of relationships,
- $\Gamma:R \Rightarrow E\times E$ is a function that maps elements to elements via a relationship. $\Gamma:R \Rightarrow E\times E$ defines multiple relations between each element. The relations r_i and r_j are multiple relations on the same elements if $\Gamma(r_i) = \Gamma(r_j)$.

Each *element, e is defined by a finite set of properties* $_i$ $\{p_1, p_2 ... p_k\}$, such that each element has a finite set of properties. Likewise each relation, *r is defined by a finite set of properties* $\{p_1, p_2, ... p_k\}$. Each property, p_i is an ordered pair that defines a name and a value (e.g., {(*type*, *Class*), (*name*, "*stack*")…}).

The model consists of a finite set of elements *E* and a finite set of relationships *R*. The set *E* contains instances of all metamodel elements such as class, namespace, package, component etc., and the set *R* provides all instances of relationships including association, generalization, dependency, etc. These correspond to the meta-classes defined in the UML metamodel. As can be seen from the above definition elements and relationships are both first class entities.

The elements and relations are mapped with the function, Γ. Given a relationship r_i, the mapping Γ tells which elements are its end points.

A property of a member of the set *E* could be the type of element such as (*type*, *Class*), and a property of a member of the set *R* could be the type of the relationship such as (*type*, *Generalization*). Thus, this definition of model makes the elements and relations along with their property sets available for model slicing

The *context-free model slice*, S_{cf}, of a given UML model *M*, is defined as a function over a model and determined by the specified slicing criteria, C_{cf},

$$S_{cf}(M, C_{cf}) = M' = (E', R', \Gamma') \subseteq M$$

The *context-free slicing criteria* C_{cf} is defined as a triple of constrains that must all be satisfied to construct *M'*.

$$C_{cf} = (I, S, D)$$

The initial-element set, $I = \forall e \in E \mid P_I(M)$ specifies the initial elements of the slice. The predicate $P_I(M)$ is constructed to be satisfied for elements in the initial set.

The selected-element set, $S = \forall e \in E \mid P_S(M)$ specifies the elements selected for inclusion in the resultant slice. The predicate $P_S(M)$ is defined so that only elements of interest are selected.

The dimension-set, $D = \forall r \in R \mid (P_D(M) \wedge T(M) \wedge B(M))$ specifies the relationships of interest (a.k.a., dimensions) to be included in the slice and traversed in its computation. The predicate P_D defines which relationships are included in the slice. The predicate *T(M)* defines a terminating condition of the computation with respect to each of, or all, the relationships. The bounding predicate *B(M)* is the computational upper bound on the path length between elements with respect to each of, or all the relationships, of the slice.

Consider a hypothetical UML class model with classes and relations. To extract a subset of the model, with a class named *ClassA* and all its base classes involved in a generalization relation up to level 2, the predicates for the slicing criteria are specified as follows,

$P_I(M) := e \supseteq \{(name, \text{"ClassA"})\}$
$P_S(M) := T$
$P_D(M) := r \supseteq \{(type, \text{"Generalization"})\}$
$T(M) := F$
$B(M) := \forall r \forall e \exists e' \in E \mid \Gamma(r) \equiv (e, e')$
$\wedge |path(e, e', r)| \leq 2$

The computation of a slice proceeds along a given dimension (i.e., relationship type) of the multi-graph, injecting elements into the model slice according to the specification in the slicing criteria. All three predicates must be satisfied together, that is $I \wedge S \wedge D$ must hold true for all e_i and r_i in *M'*. The computation of the slice ends when one of two conditions is met: the termination condition for the computation or the bounding conditions for each dimension. For example, the computation could end if a class of a certain stereotype or name is encountered (the first condition is met) or there are no more elements to be found along a dimension's path (the second condition is met). Furthermore, the computation can be bounded by the length of the path traversed in each dimension.

2.2. Context-free Model Slicing Algorithm

The algorithm given in Figure 1 computes a context-free model slice and starts with the computation of initial elements set E_I from the specification P_I. All the members of set E_I are included in the element set *E'* of the slice. The computation of the slice considers one element of the set E_I at a time for traversal. The algorithm proceeds in a breath-first (i.e., level order) traversal of the model with the initial element e_i considered to be at level 0. For each element at a level, all the elements at the next level involved in relations (satisfying the dimension predicates $P_D \wedge B(M) \wedge \neg T(M)$) are considered as candidates for inclusion in the slice. All the elements satisfying the element selection predicate P_S and the homogenous path criteria (further discussed in property 10 of section 2.3) are included in the slice Also, the relation between the element at the next level satisfying the above conditions and the element under consideration at a current level, is included in the slice. After all the relations and the elements involved at the current level are considered, the algorithm moves to the next level. The level traversal terminates when the terminating predicate *T(M)* is met or the upper bound specified by the bounding predicate *B(M)* is achieved. All the above steps are repeated for each element in the initial element set and a final slice S_{cf} is obtained on completion.

```
Procedure $S_{cf}(M, C_{cf})$
$E' = \varnothing$
$R' = \varnothing$
$S_{cf} = (E', R')$
$E_I = \{e_i | \forall e_i \in E\ [P_I(M)]\}$ ; compute initial elements set
$E_I \subseteq E'$; include the initial set in the slice
for each $e_i \in E_I \wedge e_i \neq \varnothing$ ; compute slice for each initial element
        $l = 0$ ; start with the level 0
        $e_l = e_i$ ; only element at level 0
        do BFT of $M$ starting at $e_i$ ; perform breadth first traversal of the model
                $(E_{l+1}, R_{l+1}) = \{e_{l+1} \in E, r_{l+1} \in R \mid (r_{l+1} \in D \wedge \Gamma(r_{l+1}) \equiv (e_l, e_{l+1}) \wedge Hpath(e_i, e_{l+1}, r_{l+1}) \wedge P_S(M))\}$ ; all elements at the next level and relations satisfying the dimension relationships, inclusion condition, and having homogenous path
                        $E_{l+1} \subseteq E'$; include the elements
                        , $R_{l+1} \subseteq R'$; include the relations
        $l = l+1$; go to the next level
return $S_{cf}$
```

Figure 1. Context-free model slicing algorithm

2.3. Properties of a Model Slice

The following properties are derived from the definition of a model slice, S_{cf}:

1. A slice may contain gaps in the path along a dimension. The computation of the slice does not stop if it encounters an element not satisfying the predicate P_S; rather the computation continues in the specified dimension, excluding the elements that fail to satisfy the selection criteria..,
2. A tautology for the selection predicate P_S indicates all the elements along the dimensions satisfying the dimension predicate, P_D are included in the model slice.
3. If the initial element predicate, P_I, is equivalent to the selection predicate P_S, the resultant slice is identical to the initial element set.
4. Every element included in the model slice must satisfy one or both of the predicates, P_I or P_S. This gives the following invariant of the model slice,
   $\forall e \in E' \mid (P_I(M) \vee P_S(M))$
5. There is no explicit specification of directionality in the dimension predicate P_D. Directionality of a dimension is implicitly defined by the semantics of each relationship in the UML metamodel. For example the metamodel defines generalization and specialization to be separate dimensions of the same inheritance relationship. Therefore, model slicing views generalization and specialization as different dimensions.
6. A tautology for the dimension predicate P_D indicates all the relations are of interest irrespective of their properties such as role or stereotype information and the entire model needs to be traversed to compute the model slice. In this case, the selection predicate, P_S, will be evaluated for every element reachable along a path from elements in the initial element set, E_I.
7. The dimensions given by the dimension predicate, P_D are specified only with respect to the initial elements satisfying the initial element predicate, P_I. Thus, the elements included in the slice through the satisfaction of P_S are not recursively traversed for all relations of interest but only for those of the initial elements. For example, suppose that a derived class is involved in only generalization while the immediate base class is involved in generalization and association. If the derived class is the initial element than a slice, with respect to entire association and generalization relations, involves only traversal of the generalization relation of the base as the derived class is not involved in any associations. This implies another implicit condition for an element to be included in the slice. Not only is the element required to satisfy the selection predicate, P_S, but there must be a homogenous path (i.e., a path in the same dimension) from the initial element to elements included during traversal.

However, the resultant slice may not contain all elements along the path (i.e., it has gaps). The homogenous path condition (*Hpath*) is stated below,
Hpath(e_i, e_j, *r*) = $\forall m \mid \Gamma(r) \equiv (e_m, e_{m+1})$ *where*
$i <= m < j$
e_i *is the initial element*
e_j *is the element considered to be included in the slice*

8. Each kind of relation in the dimension specification is allowed to have a different terminating predicate ,*T(M)* and bounding *predicate, B(M).*
9. The literal value *F,* for *the terminating predicate, T(M)* indicates that there is no termination condition for the slice and computation for slice must continue until the bounding predicate, *B(M)* is satisfied for all relations.
10. The traversal of each element satisfying the initial element predicate, P_I occurs in a breadth-first manner. The traversal fans out in levels, and all the relations and elements at a level are considered before moving to the next level.
11. A tautology for the bounding predicate, *B(M)* specifies the traversal along the dimension relationship(s) terminates if there exist no more elements along that path (e.g., reaching the root of an inheritance hierarchy while traversing a generalization dimension).
$\exists \neg$ *(path(ei, e, r) > path(ei, ej, r)) where*
elements e_i, e_j *are already visited and*
element e is not visited along a relation r
It is possible and valid for the final slice to have elements (relations) that do not satisfy the terminating predicate *T(M)*. The elements (relations) to be included in the slice are only governed by the selection predicate P_S (P_D). The slice may terminate early because of the bounding predicate *B(M)* is met before the terminating predicate *T(M)*. .

3. Applying Model Slicing

In this section, we demonstrate by example the ability of model-slicing to satisfy questions that can be asked of UML class models during software maintenance. Three applications are given dealing with design understanding, fault location, and metric relevance. These applications of model slicing are relatively simple to give the reader an understanding of our method and a feeling of how it can be used to assist in addressing particular tasks. Larger applications of model slicing require sophisticated tool support to formulate the criteria and are beyond the scope of this presentation.

These examples were constructed by hand and verified with a proof-of-concept implementation of the model slicing algorithm. All the applications presented here operate on a portion of the UML metamodel. This is a domain many readers are familiar with and should enable them to follow along without lengthy explanation of the domain.

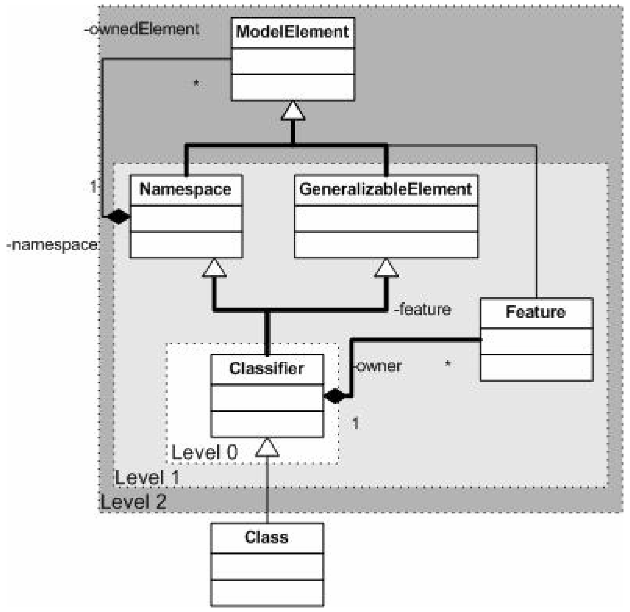

Figure 2. The model slice is shown within the grey boxes. The levels reflect the traversal of two relationships and represent the iterations of the slicing algorithm. The class *Class* is not included in the slice.

3.1. Design Understanding

<u>Question:</u> *How can a programmer discover relationships between a specific class and other classes in a UML system model?*

Programmers, when faced with such problems, might typically browse through project software artifacts including reference manuals, UML class diagrams, and source code to discover relationships between one class and its associates (base classes, aggregations, dependencies, etc.). Rarely, even in good software documentation, is this information localized for easy consumption. Model slicing can be used as a query mechanism to provide concise views of the programmer's informational needs. Consider a snippet of the UML metamodel shown in Figure 2. A programmer looking for the inheritance hierarchy and immediate associations of the *Classifier* class could use the following slicing criteria to determine related classes.

$P_I(M) := e \supseteq \{(name, \text{" Classifier"})\}$
$P_S(M) := T$
$P_D(M) := r \supseteq [R1 \vee R2]$
$R1 = \{(type, \text{"Generalization"})\}$
$R2 = \{(type, \text{"Association"})\}$

$T(M) := F$
$B(M) := (\forall r \supseteq R1) \vee$
$(\forall r \supseteq R2 \wedge (\forall e\ \exists e' \in E\ |path(e, e', r)| \leq 1))$

The computation of the model slice begins with the computation of the initial element set, E_I, from the specification, P_I, and results in only one class, *Classifier*. The two relationships computed from the predicate P_D are *Generalization* and *Association*. The predicate *B(M)* indicates the bounding condition for the traversal of these relationships. The *Association* relationship is bounded at the first level, whereas the *Generalization* relationship is bounded only when the traversal has exhausted all the elements in the *Generalization* path (refer to item 11 of slice characteristics in Section 2.3). The terminating condition, *T(M)*, is assigned the value *F* to indicate no explicit termination condition.

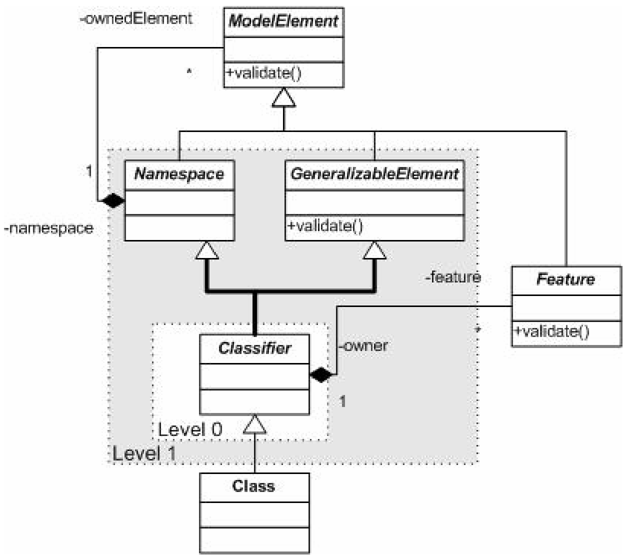

Figure 3. The model slice is contained in the gray box. The slice reflects the first base class implementing the *validate* method.

Before executing the breadth-first-traversal in the slicing algorithm, the slice consists of elements in the initial set, shown by the level zero in Figure 2. Following the algorithm given in Figure 1, the traversal starts with *Classifier* and considers elements at level one. The relationships traversed to the candidate elements are shown in bold. The algorithm selects two elements along the *Generalization* path (*Namespace* and *GeneralizableElement*) and one element along the *Association* path (*Feature*). After the first iteration, the slice now includes those elements contained in level zero and those contained in level one. In the second iteration of the algorithm, only elements involved in the *Generalization* relationship are considered because the bounding condition of the *Association* relationship was satisfied at level one. All the elements in the *Generalization* relationship at this level are included in the slice and are shown to be contained in level 2. The computation of the model slice terminates after this iteration as there are no more elements in the *Generalization* path. The final slice is depicted in Figure 2, consisting of all elements contained within the outermost level.

The class, *Feature*, is included as part of *Association* because aggregation is a kind of association in the UML metamodel. Note the *Association* computation is not transitive as only the immediate associations of *Classifier* are considered (i.e., *Feature*). Associations of base classes are not considered for the slice. If those associations are of interest, the base classes must be included in the initial element set of the slicing criteria.

3.2. Fault Localization

__Question:__ *How can a programmer find classes that implement a specifically named method, in one of which a bug is known to exist?*

In certain application domains such as user interface frameworks, it is common for overloaded methods in derived classes to call the same methods of their base classes. In this example, we consider a similar call chain that is used to validate a model against the UML well-formed rules and other OCL constraints. As shown in Figure 3, classes requiring additional validation overload the *validate()* method to check metaclass-specific features.. Calculating a model slice for the entire inheritance hierarchy may result in too many classes. In this case, the programmer would want to refine the slicing criteria to return a smaller set of elements relevant to their query. This gives the programmer options when composing the criteria. Should the slicing algorithm terminate on the first class that implements method or include all classes that implement the method? This example considers two cases:

1) Include all classes in the inheritance hierarchy and stop when the first base class implementing the method is found and

2) Include only classes that implement the method and traverse the entire inheritance hierarchy.

The slicing criteria for the first case are given below:

$P_I(M) := e \supseteq \{(name, \text{" Classifier"})\}$
$P_S(M) := T$
$P_D(M) := r \supseteq \{(type, \text{"Generalization"})\}$
$T(M) := e_t \supseteq \{(implements, \text{"validate"})\}$
$B(M) := T$

The specification states that the algorithm starts with *Classifier* (i.e., P_I specification) and traverses the *Generalization* path including all the classes found during this traversal in the model slice until the first class implementing the specified method (i.e., the termination predicate is met) is found. Here, the value of *B(M)* indicates the worst case situation in which the entire *Generalization* path must be traversed to find candidate elements for the slice. The slice computation terminates when the first element (i.e., base class) implementing the method named *validate()* is found, or the path is exhausted. The selection predicate, P_S, indicates that all candidate elements are included in the slice. Thus the slice will contain classes on the traversal path that do not actually implement the specified method. The final slice, obtained by applying the model slicing algorithm, is shown in Figure 3 and consists of all elements contained within the outermost level.

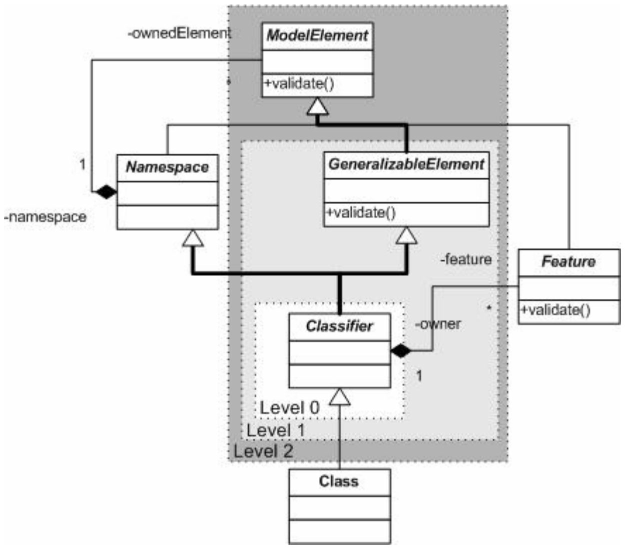

Figure 4. The gray boxes (each level representing iterations of the algorithm) gives the model slice after searching for all base classes implementing the *validate* method.

Notice the base class *Namespace* is included in the slice. Because *Namespace* and *GeneralizableElement* are at the same level in the traversal, the inclusion of *Namespace* depends on the specific traversal order of classes at this level. If *Namespace* occurs before *GeneralizableElement* then it is included in the slice. Here, we assume the order of traversal is from left to right so *Namespace* is included in the final slice.

Now consider the slicing criteria for the second case:

$P_I(M) := e \supseteq \{(name, \text{" Classifier"})\}$
$P_S(M) := e_s \supseteq \{(implements, \text{" validate"})\}$
$P_D(M) := r \supseteq \{(type, \text{"Generalization"})\}$
$T(M) := T$
$B(M) := T$

In this case, the only classes included in the slice are those that implement the method (i.e., determined by the selection predicate P_S). Also, the slice computation continues until the entire *Generalization* path is exhausted. The final slice is shown in Figure 4, consisting of elements contained within the outermost level. Note that the path from *Classifier* to *Namespace* is represented in bold. This indicates that the algorithm actually followed the path, visiting the *Namespace* class. However, the selection criteria are not met and so the class is not included in the final slice.

3.3. Metric Relevance Slicing

Question: *How can a software engineer find classes in a software design by metric analysis to identify potential refactorings?*

The ability of a software engineer to locate and analyze classes for potential refactorings is critical to reengineering activities. The engineer must know the design relevant to the classes targeted for refactoring and the impact the refactoring may have on related classes. As shown, model slicing has the ability to extract regions of interest from a UML model. However, in this example, we augment a typical UML model with LCOM (lack of cohesion metric) [4] values in order to provide additional analytic functionality in the slicing criteria and therefore, the ability to answer questions of this nature.

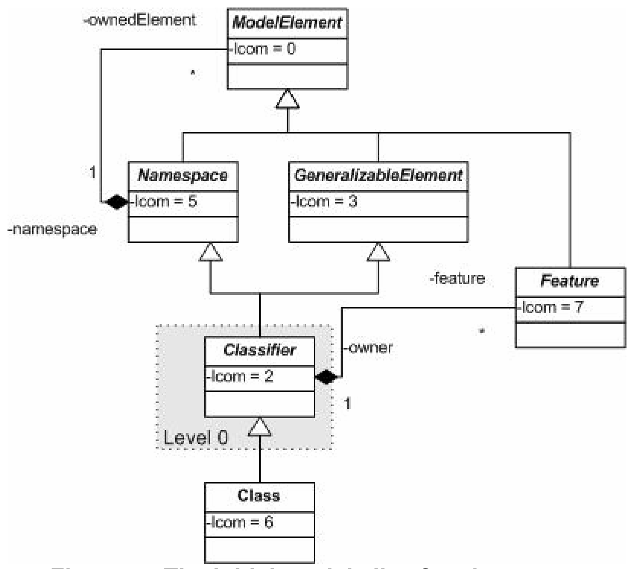

Figure 5. The initial model slice for classes augmented with LCOM values of 2 or greater is shown as a gray box.

Figure 5 shows a model augmented with LCOM values. These values are simply treated as attributes of classes in which they appear. The following slicing criteria obtain a slice of classes with LCOM values greater than 2 in the inclusive inheritance hierarchy of elements in the initial set (i.e., the *Generalization* and *Specialization* relationships are both traversed).

$P_I(M) := e \supseteq \{(LCOM, 2)\}$
$P_S(M) := e \supseteq \{(LCOM, gte(2))\}$
$P_D(M) := r \supseteq [R1 \vee R2]$
$R1 = \{(type, \text{“Generalization”})\}$
$R2 = \{(type, \text{“Specialization”})\}$
$T(M) := T$
$B(M) := T$

Note the specification of the selection predicate, P_S, includes an evaluation function, *gte*, which is responsible for evaluating the numeric comparison between the elements LCOM value and the constant value 2.

The initial model slice is show in Figure 5, containing only *Classifier*. Had there been other classes in the model with LCOM values of 2, they would also be included in the initial element set. After computing the slice, the result is shown in Figure 6, consisting of all elements contained within the outermost level.

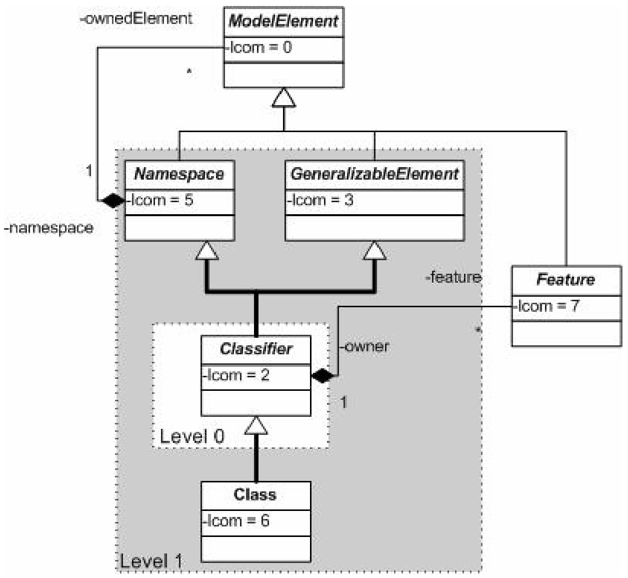

Figure 6. The final model slice for classes with augmented LCOM values greater than or equal to 2.

4. Related Work

A variety of artifacts are produced or derived throughout the software development life cycle. These artifacts may include design models such as UML class models, source code, and additional models representing various aspects of the software at different abstraction levels. Most maintenance or evolution tasks (e.g., corrective maintenance, perfective maintenance) require only the most relevant parts of these artifacts. However, extracting this subset may utilize information from other artifacts or their derived abstractions. The ability to extract relevant subsets of artifacts is typically a prerequisite for answering many types of questions about the artifacts and understanding their contents. Additionally, these subsets are often utilized to construct models with higher levels of abstraction.

In this section, we examine existing techniques for extracting relevant subsets of software artifacts at the source code and UML design levels. We classify and discuss proposed or available methods for subset extraction/identification, based on their targeted abstraction (i.e., the artifact itself or a derived abstraction of the artifact such as a control flow graph).

4.1. Program Slicing

Most related to our work presented in this paper is program slicing. It is a technique to extract a subset of a program. Source code models such as data flow, control flow and dependency graphs are utilized in computing program slices [8, 20]. Thus, program slicing is an effective mechanism to extract source code parts by utilizing information available only at the source code level. Program slicing techniques for object-oriented software is described in [11]. In this work, the program is represented in the form of the system dependency graph (SDG). Here, SDG is extended to represent class, inheritance, interactions, and polymorphism extracted from the source code. The slicing algorithm is then applied on SDG with the slicing criteria modified to allow for slicing based on data member variable or method of a class. However, the granularity of slice is at a statement-level.

In [17] the combination of statement-level slicing within a class and a class level slicing for a collection of classes along an inheritance relation (i.e., class hierarchy) is considered within a context of a usage of classes. The desired slice is the one with only classes and statements within classes relevant to its usage in a particular context. Program slicing has demonstrated its applicability in debugging, testing, differencing source code versions, integration of software modules [8], impact analysis [6, 18], and other software engineering applications.

However, program slicing is not applicable to UML design models as it is only capable of addressing the source code, not higher levels of abstraction such UML models or other derived abstractions. Without further extension and refinement, program slicing has limited capabilities in these domains.

It should be noted that other approaches such as call graph analysis, concept analysis, and clustering have rarely, if ever, been applied with regards to these higher level abstractions, particularly UML models.

4.2. Querying UML Models

Just as there are methods to query and extract information from source code, a variety of tools and techniques have been proposed to perform similar activities on UML models. In this subsection, we discuss existing methods to query and extract information from UML class models – the subset of a model that deals with the structural design of the system (i.e., classes, attributes, and operations). These approaches include constraint and query languages, XML processing, and model processing.

The UML object constraint language (OCL) [1, 7, 19] allows querying of UML models. It is primarily used for model validation and constraint checking. Recent work in Model Driven Architecture (MDA) has proposed using OCL as a query language to satisfy the query component to the QVT (Query/View/Transformation) specifications [2]. As a query language, OCL can be used to extract model elements that satisfy some condition (e.g., all abstract classes in a model).

Many approaches to querying UML models involve analysis or operations on the XML-based interchange format for UML models, XMI [10, 13, 16]. These approaches apply XML processing techniques such as XPath, XQuery and XSLT to extract data from the XMI files. This approach is similar to those used in XML-based source code or AST analysis.

Other approaches involve model analysis in the context of additional semantic information. In [9] a metric-based approach is proposed to derive subparts of UML class diagrams. A high level view of the subparts exhibiting a particular metric-based feature such as coupling (referred to as a coupling diagram) is obtained, and then classes within a particular range of metric values are extracted and visualized via a diagram. This approach is particularly good at "pruning" large UML diagrams to show only the relevant classes in the diagram. However, this approach is very limited in the types of pruning that can be done.

Other approaches involve the use of additional languages and technologies to query or validate UML models. In [12], OCL expressions are translated to SQL statements to query and evaluate models stored in a relational database. In [15], Python and OCL are used together to provide more procedural control for the evaluation of such queries.

Concept location, a human intensive activity is used to search for and find classes in the UML class diagrams [14] and/or units of source code. These approaches use the Abstract System Dependence Graph (ASDG) [3] of classes that are affected because of a problem domain level concept or feature change. In this approach a user decides the starting component, next related component to visit, and inclusion candidates in the list of affected components. This can be used to locate features in the design or source code level.

The concept analysis and cluster analysis techniques can be used in conjunction with model slicing to identify concepts and clusters in the UML models. The model slices provides an alternative approach to obtaining context information needed for concept analysis. The criteria used to compute model slices can also be used as a basis for clustering criteria. Moreover, model slicing can be used to automate a great deal of the human intensive components of concept location. The selective representation problem in [9] extracts a higher level view (i.e., a coupling diagram) of the model based on the desired metric value such as CBO [5] to select a subset of classes. However, model slicing requires no such higher level views as such information is considered as part of the model itself.

5. Conclusions & Future Work

In this paper we introduced the concept of model slicing pertaining to UML class models. The work generalizes the concept of program slicing so that it can be applied to more abstract models. The ultimate goal is to provide an automatic mechanism that will enable developers to extract task-specific UML sub-models by giving specification in terms of UML-level constructs. This type of approach will support the development of sophisticated tools to automatically extract meaningful sub-models of large system design model so they can be visualized or analyzed to facilitate maintenance and evolution tasks.

Three applications are given to demonstrate the usefulness of model slicing for the purpose of obtaining information about a class model. Current techniques to obtain similar information from class models consist either of manual inspection or very specialized one-time solutions that can not be applied broadly. Our approach addresses the problem of extracting relevant subsets of a class model in a very general manner. While these examples may be limited in their scope, it is conceivable that the underlying capabilities of model slicing can support a much wider range of software engineering tasks such as impact analysis with respect to a design model.

Model slicing is realized as a set of predicates that specify a slicing criterion. As can be seen, even in simple examples, this specification can be difficult to articulate. We envision that languages such as OCL can be used to implement the predicates required to compute the model slices.

In this vein, we are extending our prototype implementation of the model slicing algorithm and plan to apply it to large models of real systems (e.g., an open source systems such as HippoDraw). The tool is built using our in-house developed Open Modeling Framework (OMF) and the Python scripting language. OMF is a framework for the storage and representation of UML models. Alternatively, one can also build an implementation on top of Rational Rose as a plugin using VBScripts. In the future we plan to further investigate the usefulness of UML model slicing through a set of question-and-answer experiments with existing software systems.

We are actively investigating the concept of context-sensitive slicing of UML models using a behavioral model, such as a sequence diagram that is implicitly linked to the static class model. This will result in a slice of the class model in the context of the modeled behavior. We feel this has much promise in producing relevant sub-models. However, the work presented here is a necessary prerequisite to realizing a context-sensitive slice.

6. Acknowledgements

This research was supported, in part, by a grant from the National Science Foundation (CCR-02-04175).

7. References

[1] Akehurst, D. H. and Bordbar, B., "On Querying UML Data Models with OCL", in Proceedings Fourth International Conference on the Unified Modeling Languages (UML'01), Toronto, Canada, October 1-5 2001, pp. 91-103.

[2] Appukkutan, B., Tratt, L., Clark, T., Reddy, S., Venkatesh, R., Evans, A., Maskeri, G., Sammut, P., and Willans, J., "QVT-Partners Revised Submission to MOF 2.0 Query/View/Transformations RFP", Object Management Group, Document ad/03-08-08, August 2003.

[3] Chen, K. and Rajlich, V., "Case Study of Feature Location Using Dependece Graph", in Proceedings 8th International Workshop on Program Comprehension (IWPC'00), Limerick, Ireland, June 2000 2000, pp.

[4] Chidamber, S. R. and Kemerer, C. F., "Towards a Metrics Suite for Object Oriented Design", in Proceedings OOPSLA'91, 1991, pp. 197-211.

[5] Chidamber, S. R. and Kemerer, C. F., "A Metrics Suite for Object Oriented Design", IEEE Transactions on Software Engineering, 20, 6, 1994, pp. 476-493.

[6] Gallagher, K. and Lyle, J., "Using Program Slicing in Software Maintenance", Transactions on Software Engineering, 17, 8, August 1991 1991, pp. 751-762.

[7] Gogolla, M. and Richters, M., "On Constraints and Queries in UML", in Proceedings Workshop on the Unified Modeling Language - Technical Aspects and Applications, Mannheim, Germany, November 10-11 1997, pp. 109-121.

[8] Horwitz, S. and Reps, T. W., "The Use of Program Dependence Graphs in Software Engineering", in Proceedings International Conference on Software Engineering (ICSE), Melbourne, Australia, May 11 - 15 1992, pp. 392 - 411.

[9] Kollmann, R. and Gogolla, M., "Metric-Based Selective Representation of UML Diagrams", in Proceedings Sixth European Conference on Software Maintenance and Reengineering(CSMR'02), Budapest, Hungary, March 11 - 13 2002, pp. 89-98.

[10] Kurtev, I. and van der Berg, K., "Model Driven Architecture Based XML Processing", in Proceedings ACM Symposium on Document Engineering (DOCENG'03), Grenoble, France, 2003, pp. 246-248.

[11] Larsen, L. and Harrold, M. J., "Slicing object-oriented software", in Proceedings Proceedings of the 18th international conference on Software engineering (ICSE96), Berlin, Germany, March 25 -29, 1996 1996, pp. 495 - 505.

[12] Marder, U., Ritther, N., and Steiert, H.-P., "A DBMS-based Approach for Automatic Checking of OCL Constraints", in Proceedings OOPSLA'99 Workshop on Rigourous Modeling and Analysis with the UML: Challenges and Limitations, Denver, Colorado, November 1-5 1999, pp.

[13] Peltier, M., Bézivin, J., and Guillaume, G., "MTRANS: A general framework, based on XSLT, for model transformations", in Proceedings ETAPS'01 Workshop on Transformations in UML, Genova, Italy, April 7 2001, pp.

[14] Rajlich, V. and Wilde, N., "The Role of Concepts in Program Comprehension", in Proceedings International Workshop on Program Comprehension (IWPC 2002), Paris, France, June 27 - 29 2002, pp. 271-278.

[15] Siikarla, M., Peltonen, J., and Selonen, P., "Combining OCL and Programming Languages for UML Model Processing", in Proceedings UML'03 Workshop on OCL 2.0 - Industry Standard or Scientific Playground?, San Francisco, California, October 21 2003, pp.

[16] Stevens, P., "Small-Scale XMI Programming: A Revolution in UML Tool Use?" Automated Software Engineering, 10, 1, January 2003 2003, pp. 7-21.

[17] Tip, F., Cho, J. D., Field, J., and Ramalingam, G., "Slicing Class Hierarchies in C++", in Proceedings Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA96), San Jose, California, United States, October 06 - 10, 1996 1996, pp. 179 - 197.

[18] Tonella, P., "Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis", Transactions on Software Engineering, 29, 6, June 2003 2003, pp. 495-509.

[19] Warmer, J. and Kleppe, A., The Object Constraint Language : Precise Modeling with UML, 1st ed., Addison-Wesley Pub Co, 1998.

[20] Weiser, M., "Program slicing", in Proceedings International Conference on Software Engineering (ICSE'81), San Diego, California, March 09 - 12 1981, pp. 439 - 449.