An Approach to Pruning Metamodels like UML

Zhiyi Ma
Software Institute, School of Electronics Engineering and Computer Science, Peking University, Beijing, China
Key Laboratory of High Confidence Software Technologies, Peking University, Ministry of Education, Beijing, China

Keywords: Metamodel, Pruning Algorithm.

Abstract: There are a large number of modeling languages based on metamodels, and many of the languages are large and complex. In many cases, only part of a metamodel is needed. Hence, it is necessary to automatically extract needed part from a metamodel. By deeply analyzing the characteristics such as special relations between packages and step-by-step strictly defining mechanism of modeling concepts, this paper presents an approach to pruning metamodels like UML as needed. The approach can effectively prune metamodels, control the size of pruned metamodels, and make pruned metamodels comply with its initial metamodels.

1 INTRODUCTION

With the increase of the scale and complexity of software, models become important artifacts in software development nowadays. Such models usually are built in modeling languages.

A metamodel is a model of models (OMG, 2013), i.e. a model that specifies the language for building models. In this paper, metamodels are the ones like UML. That is, a metamodel describes the abstract syntax and the static semantic meaning of a modeling language with meta-class diagrams and object constraint languages, respectively (OMG, 2011b; OMG, 2011c; Flatscher, 2002). The abstract syntax defines a set of modeling concepts, their attributes, relationships between them, and the rules for combining these concepts to build partial or complete models; the static semantic meaning typically specifies queries or invariant conditions that must hold when applying a metamodel, e.g. UML OCL expressions.

The core of many modeling languages is a metamodel, with a detailed explanation (text description in most cases) of the semantics of each modeling concept, and such languages are called metamodel-based modeling languages. The organizations such as International Organization for Standardization (ISO) and Object Management Group (OMG) have released a large number of metamodel-based modeling languages, e.g. OMG has released more than 200 such specifications, and many of such specifications have more than one version, e.g. there are 14 versions of UMLs. With the development of society, new metamodels will come forth. In addition, many of the metamodels are built upon existing metamodels using additional techniques such as profiling and package merge, and tend to become bigger and bigger (Frédéric et al., 2013).

Many metamodel-based modeling languages are large in size and complex in structure. For example, the number of the pages of the core part of UML 2.4.1(OMG, 2011a; OMG, 2011b) is more than 1000, its meta-classes are more than 400, and a meta-class may have many properties. Besides, there are complex dependency relationships between the meta-classes. The Common Warehouse Metamodel (CWM) 1.1 is 600 pages long even without its Core package. SysML is a profile of UML and nearly 300 pages long without the part of UML specification.

The size and complexity of such metamodels make it is extremely difficult for language builders and tool developers etc. to fully identify the dependencies among concepts and to determine whether the metamodels capture all required dependencies (Frédéric, 2013; Robert, 2007). Moreover, in many cases application modelers only need to understand and apply parts of metamodels, not whole.

It is difficult for users, such as application modelers, transformation rule developers, and modeling tool developers, to directly study and apply the metamodels of large and complex modeling languages. The novices usually first study

Ma Z.
An Approach to Pruning Metamodels like UML.
DOI: 10.5220/0006144004090417
In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*, pages 409-417
ISBN: 978-989-758-210-3

the basic part of the metamodels, and then others. That is, to learn a language, one should follow the principle of making gradual and orderly progress.

With the evolvement of modeling languages (such as UML), it is a trend that one defines metamodels with more and more multiple inheritance and deeper and deeper inheritance hierarchy (Brian, 2005). The trend increases the difficulty for users to learn and apply the modeling languages (Dori, 2002). For technical experts, they also often need to extract information on given elements of the languages. For example, which elements are used to define given elements, and which elements are defined by given elements. Therefore, it is necessary to extract needed parts from the metamodels of the modeling languages.

Software models are usually built in general-purpose modeling languages (such as UML), but such modeling languages cannot satisfy modeling requirements of many fields. Therefore, the modeling languages need to be extended. For example, OMG has released many UML profiles. Moreover, the modeling languages will constantly evolve, with the development of business fields and software development technologies. The evolution means to change (i.e. add, delete, and modify) constructs of the modeling languages. Therefore, it is necessary to fix the range of influence of an expansion and a change by calculating the elements whose definitions are related to the extension and change (Jiang, 2004). If the work is made manually, it is tedious and error prone (Rober et al., 2007). The work can be finished by extracting needed parts from metamodels.

Model transformation, which transforms source models into target models, is an important way to develop applications. Two kinds of models are usually built by using metamodels. Actually, only parts of such metamodels are usually used. For example, in object-oriented development, transforming persistent classes and relations between them into tables described in CWM only uses part of UML's Classes package and part of CWM's Record package. This means transformation rule developers only need to study and apply needed parts. This shows that extracting needed parts from metamodels avails not only modeling but also model transformation.

As mentioned above, modeling languages and transformation languages will constantly evolve. For maintaining existing models, it is usually necessary to find which modeling elements used to build given models are affected by the changed elements of the languages and then to modify the models according to the modeling elements. The first work can be completed by extracting needed parts from metamodels according to evolved elements. Similar work is for maintaining transformations.

The quality of metamodels is very important since one uses them to build models. Indeed, there are defects in many metamodels (Brian, 2005; Ma et al., 2013). An approach to assure the quality of metamodels is using divide and conquer strategy, i.e. extracting needed part around each of the subjects of the metamodels and inspecting it.

The above analysis shows that, for large and complex modeling languages, in many cases it is necessary to extract needed parts from their metamodels, namely pruning metamodels here.

It is extremely difficult to manually prune such metamodels. A solution is automatically pruning metamodels with tools, which may be built based on the modeling tools that have encoded metamodels, for example, Rational Rose and Eclipse UML 2. The tools for pruning metamodels must support a calculation that can decide which elements are necessary.

The existing approaches for pruning class models are not applicable to prune metamodels because metamodels have their own characteristics (see Section 2). There is some work on pruning a metamodel, but such work even has nothing to do with the architecture and some of the important characteristics of a metamodel. Therefore, it is necessary to present a new approach to automatically pruning metamodels.

By deeply analyzing the characteristics of metamodels such as the special relations between packages and the step-by-step strictly defining mechanism of modeling concepts, this paper presents an approach to pruning metamodels.

The structure of the paper is as follows. Taking the case of UML 2.4.1, Section 2 presents an approach to pruning metamodels; Section 3 discusses the approach; Section 4 analyzes related work. Finally, conclusions are drawn.

2 CALCULATION METHOD

In metamodels, meta-classes are defined in packages, and some of them are defined step-by-step in different packages. For example, meta-class Classifier first appears in Infrastructure::Core::Abstraction::Classifier, and then is further defined in Infrastructure::Core::Constructs via an import relation. Therefore, we need to input the specified

meta-classes and the packages at which the meta-classes locate when pruning metamodels. They form a starting point set for pruning a metamodel.

Packages are usually applied to control the complexity of large metamodels, and thus the size of a class diagram in a package is not large and it is unnecessary to limit the relation path length between meta-classes in a package. However, the dependent path length between packages is considerable (for example, the maximum dependent path length between packages in UML2.4.1 is 7), and thus the length may be limited to get a smaller size metamodel. For example, one usually learns a concept from the near to the distant around it. For a given meta-class, if the path length from the package including it to the related packages is not specified, the default value is the maximum dependent path length from the package which it locates at to the related packages, here we mark the default value is @; if the length is 0, this means that the metamodel in the package at which the given meta-class locates is only calculated.

In some cases, it is necessary for users to specify which packages, meta-classes, and properties of the meta-classes are undesired. For example, business process modeling does not need the State Machines package and Components package of UML, and sometime not modeling elements such as ActionInputPin and OutputPin in UML::Activities package.

The desired limit on dependent path length between packages, and the undesired packages, meta-classes, and properties of the meta-classes are the optional parameters for pruning metamodels.

A metamodel is pruned according to a starting point set and optional parameters, see Figure 1.

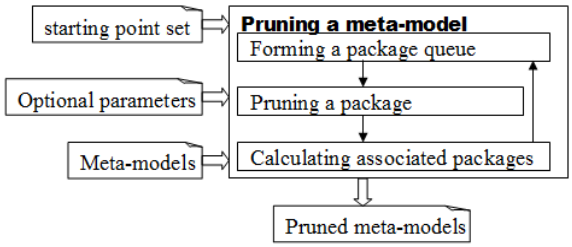

Figure 1: Overview of the pruning method.

The following are the steps of pruning a metamodel. First, packages at which specified meta-classes locate are processed in an arbitrary order to form a package queue, since our calculation method is independent of the order of calculating packages. Then each package in the queue is pruned, and the packages that relate to each package in the queue are added into the queue and recursively calculated. In the calculation, it is necessary to record the visited packages, relations between the packages, and the pruned class diagrams in these packages. Finally, the recorded elements are the output of pruning the metamodel.

Pruning an innermost package, which only includes class diagrams, means extracting the meta-classes associated with specified meta-classes. The paper calls the pruning algorithm as the pruning single package algorithm.

The following discusses how to calculate the associated packages. Some meta-classes in a package may depend on meta-classes in other packages, that is, the package may import and merge other packages. These meta-classes are grouped according to each package which they depend on, respectively, and the meta-classes in each group are taken as input to calculate the related package, respectively. The calculating results may still depend on the meta-classes in other packages, thus the calculation above is continued until the packages that do not depend on other packages or whose dependent path length is more than the specified value.

For a processed package, if it is calculated again, the results of two calculations need to be merged. For example, taking package Kernel in UML Superstructure as input for pruning the metamodel, PrimitiveTypes package is calculated twice according to the dependency relations and the two calculating results are usually different, thus it is necessary to merge the two results.

For the given meta-classes and packages as pruning inputs, we can calculate the elements that define them and the elements which they define, since metamodels describe how to define metaclasses with other ones. We refer the first calculation as backward pruning and the second as forward pruning.

We first discuss backward pruning, then explain how to do forward pruning based on the backward pruning method.

2.1 Pruning Single Package Algorithm

A package may include more than one class diagram. The paper merges the class diagrams in a package into one for the sake of the convenient calculation since many object-oriented modeling tools can merge class diagrams.

To backward prune a package, we need to consider the following cases:

(a) For each specified meta-class from input, its

parent meta-classes are recorded first, and then the parent meta-classes of its parent meta-classes are recorded, …, in this way, until the classes without any parent meta-class.

(b) For each recorded meta-class, the meta-classes with which it directly associates first are recorded, and then the unvisited meta-classes with which each of these meta-classes directly associates are recorded, …, in this way, until the meta-classes without any associated meta-class.

(c) For each of the recorded meta-classes in (b) that have submetaclasses, because it inherits the meta-association relations of their parent meta-classes, its submetaclasses first are recorded, and then the submetaclasses of each of the submetaclasses are recorded, …, in this way, until the meta-classes without any submetaclasses.

The three cases need to be considered together in following algorithms, not just sequentially.

To deal with the cases in the pruning single package algorithm, we need three definitions and one function.

Definition 1. A class diagram G is a 2-tuple G:= (V, E) where V is a set of meta-classes, and E is a set of relations between meta-classes. The relations are divided into meta-inheritance, meta-association, and meta-combination.

Definition 2. A package P is a 5-tuple P:=<G, Creq, NCreq, NPreq, OCLExps> where Creq is a set of the specified meta-classes, NCreq is a set of the specified undesired meta-classes, NPreq is a set of the specified undesired properties of all meta-classes, and OCLExps is a set of constraint expressions by written in OCL (OMG, 2003). The default value of Creq is all meta-classes in P, and the default values of NCreq, NPreq, and OCLExps all are null.

Definition 3. If a meta-class x is a submetaclass of a meta-class y, or x can navigate to y via a meta-association or meta-combination, y is an adjacent point of x. In the latter case, if y has a submetaclass z, z is an adjacent point of x.

Function 1. OCLRelatingMetaclasses (OCLExps,v), for a given class diagram, returns all meta-classes except meta-class v in the OCL expressions that include v.

The following is algorithm for pruning a single package P.

Algorithm 1: CalcPackage(P).

1 Initialization

FOR EACH v∈P.G.V DO visited[v]:=false;

ClassSet:=Ø;

2 Calculate related meta-classes in OCL expressions of P

FOR EACH v∈P.Creq DO

ClassSet←ClassSet ∪ OCLRelatingMeta-classes (P.OCLExps, v);

P.Creq ←P.Creq ∪ClassSet;

3 Delete the specified undesired meta-classes

P.Creq ←P.Creq–P.Ncreq;

4 Recursively prune the metamodel in P, taking each needed meta-class as a starting point

FOR EACH v∈P.Creq DO

IF visited[v]=false THEN CALL Traversal (P, v)

In a meta-class diagram, there are usually meta-associations and meta-combinations, that may be unidirectional or bidirectional. For a bidirectional relation, if it has been traversed and marked in a direction, Traversal (P, v) does not traverse it from another direction since it has been marked.

Algorithm 2: Traversal(P, v).

1 Delete the specified undesired properties of meta-classes v

v.Attributes←v.Attributes – P.Npreq;

2 Mark v , i.e. v is visited.

Mark(v);

3 Get the first adjacent point of v in P

w:=FIRSTTADJ(P.G, v);

4 Recursively calculate all related meta-classes, taking v as a starting point

WHILE w≠0 DO

4.1 If w is not visited and is not an undesired meta-class, record the relation between v and w, and then calculate all related meta-classes taking w as a starting point.

IF visited[w] = false AND NOT w∈P.NCreq THEN Mark (<v, w>); Traversal (P, w);

4.2 get next adjacent point

w:=NEXTADJ(P.G, w);

2.2 Pruning Whole Metamodel Algorithm

We already know that there are 3 kinds of relations between packages. Include means that a package is included in another package, and the meaning reflects in the package path names.

Though import and merge are different relations between packages, they all means that the defining meta-classes in a package needs to use meta-classes in another package, thus the paper deals with two relations as dependency relation for pruning metamodels. For a package depended on by a set of meta-classes, these meta-classes are the input of pruning it. Such packages are added into a package queue to wait for calculation.

Since the definition of a meta-class in a package may depend on the meta-classes in other packages, it is necessary to calculate these packages if they are not undesired packages and their path lengths are not more than given maximum dependent path length.

To deal with the above cases in the pruning whole metamodel algorithm, we need one definition and six functions.

Definition 4. A metamodel MM is 2-tuple MM:=<V, E> where V is a set of packages, E is a set of dependency relations between packages. The relations are divided into include, import, and merge.

Function 2. PathLength(p) returns the maximum length of dependent paths from package p being calculated to a set of specified starting point packages, which are input for pruning.

Function 3. CalcImportingPackages(p) returns all package names that are used in the package p.

Function 4. CalcImportingElements(p, pi) returns a set that consists of all meta-classes whose package prefix name is pi in package p.

Function 5. CalcMergingPackages(p) returns the names of the packages that are direct and transitively merged by p and have the same name meta-classes with p.

Function 6. CalcMergingElements(pi) returns a set that consists of all meta-classes whose names are in package pi.

Function 7. Add(Pqueue, (pi, CalcElements(p, pi)) adds a package with a set that consists of needed meta-classes into a package queue Pqueue.

The following are the parameters of the pruning algorithm for entire metamodels. MM is a source metamodel. A package queue Pqueue is formed with the packages which the specified meta-classes locate at. NPackage is a set that is formed with the specified undesired packages. MaxLength is the specified maximum dependent path length between packages.

Algorithm 3: PruningMetamodel(MM, Pqueue, NPackage, MaxLength).

```
1 Initialize a metamodel
MMt←MM
2 Delete undesired packages
MMt←Delete(MMt, NPackage)
3 Calculate each package in Pqueue
FOR EACH p in Pqueue do
  IF PathLength(p) ≦MaxLength THEN
  3.1 Call the pruning single package algorithm and mark the pruned package
  CALL CalcPackage (p); Mark(p);
  3.2 If PathLength(p) ≠ MaxLength, calculate the dependent packages of P and the related meta-classes belonging to each of these packages. These packages are added into Pqueue, and the relations between the packages are marked.
  IF PathLength(p) ≠MaxLength
    3.2.1 For importing packages
    PDep←CalcImportingPackages(p);
    FOR EACH pi in PDep DO
      pi.Creq=alcImportingElements(p, pi);
    Add(Pqueue, pi); Mark(<p, pi>);
    3.2.2 For merging packages
    PDep←CalcMergingPackages(p);
    FOR EACH pi in PDep DO
      pi.Creq=CalcMergingElements(p, pi);
    Add(Pqueue, pi); Mark(<p, pi>);
4 Delete not marked elements
  4.1 Delete not marked packages and the relations between them
  FOR EACH p∈MMt.V DO
    IF Unmarked(p) THEN Delete(MMt, p);
  FOR EACH e∈MMt.E DO
    IF Unmarked(e) THEN Delete(MMt, e);
  4.2 Delete not marked meta-classes and the relations between them
  FOR EACH P ∈MMt.V DO
    FOR EACH v∈P.G.V DO
      IF Unmarked(v) THEN Delete(P.G, v);
    FOR EACH e∈P.G.E DO
      IF Unmarked(e) THEN Delete(P.G, e);
```

The algorithms above are applied to backward pruning, i.e. to find which elements are used to define given elements. The following discusses forward pruning, i.e. to find which elements are defined by given elements.

To forward prune a package, we need to consider the following cases:

(a) For each specified meta-class from input, its submetaclasses first are recorded, and then the submetaclasses of each of its submetaclasses are recorded, …, in this way, until the meta-classes without any submetaclass.

(b) For each recorded meta-class, the meta-classes with which it directly associates first are recorded, and then the meta-classes with which each of these meta-classes directly associates are recorded, …, in this way, until the meta-classes without any associated meta-class.

Comparing with the backward pruning, we need to redefine adjacent point in the definition 3, and change the definitions of functions 3, 4, and 5. Except for these, forward pruning algorithms are the same as backward pruning algorithms. That is, we can use backward pruning algorithms to implement forward pruning with these redefined concept and functions.

Definition 3'. If a meta-class x is a parent meta-class of a metaclass y, or x can navigate to y via a meta-association or meta-combination, y is an adjacent point of x.

Function 3'. CalcImportingPackages(p) returns the names of all packages that depend on package p.

Function 4'. CalcImportedElements(p, pi) returns a set that consists of all meta-classes whose package prefix name is p in package pi.

Function 5'. CalcMergingPackages(p) returns the names of all packages that direct and transitively merge package p and have the same name meta-classes with p.

3 DISCUSSION

First, it should be pointed out that the approach can be used to prune the other metamodels developed by using MOF, though the paper takes the case of UML in many places.

There are several application areas where the approach can make its useful contributions:

a) Gaining needed part of a metamodel for study

Using backward pruning, users such as modelers and transformation rule builders can get and study needed part of a metamodel.

b) Evolving and extending modeling languages

Even if one attribute of a meta-class (as an extension point) in a large metamodel is changed, it is usually difficult to manually fix its influence scope. Fixing the influence scope of the extension points with our forward pruning algorithms is the work of very significance for evolving and extending modeling languages.

c) Gaining needed part of a metamodel for model transformation

It is easy to get needed part with our backward pruning algorithms. The resulted part includes not only package diagrams and meta-class diagrams extracted from the metamodel but also the code file extracted from the metamodel. The reason is that a metamodel built with a metamodeling tool is stored in code files that describe the elements of metamodels and the relations between the elements, not just diagrams (OMG,2011c), and such tool supports a bidirectional mapping between the diagrams and the code files.

d) Fixing the range of influence of a change of a metamodel and inspecting models

Taking modeling elements (i.e. concrete metaclasses) used to build the existing models and transformation rules and the old and new versions of a metamodel as input of our backward algorithms, respectively, users can fix the range of the influence of the change of the metamodel by comparing the differences between two pruning results, and further inspect the influence on the models according to the modeling elements in the range.

e) Finding and handling defects

Using the divide-and-conquer strategy, we can extract part of a metamodel around a subject with the approach, and then find and handle defects, even measure the quality of the part.

The following discussions focus on other aspects of the approach.

a) Feasibility

Modularity, layering, partitioning, extensibility, and reuse are five design principles of metamodels (OMG,2011b). Such principles ensure that the built metamodels are well structured and their components such metaclasses and packages all have independence. This provides a foundation for pruning metamodels with good effects.

b) Flexibility

The approach not only can bidirectionally (i.e. forward and backward) prune metamodels, and but also has a good scalability for controlling size of pruned metamodels. To do this, users can specify the desired and undesired packages, meta-classes,

properties of the meta-classes, and a dependent path length between packages.

c) Precision and Recall

Since a metamodel describes the abstract syntax of a modeling language with meta-class diagrams and package diagrams, the key to accurately and completely extract needed part of the metamodel is parsing the relations between the elements of the metamodels, i.e. which elements are used to define given elements, and which elements are defined by given elements.

For a package, the algorithms 1 and 2 traverse metaclasses inside it according to the relations between them, i.e. generalizations and associations (including combinations), and if and only if the metaclasses between which there are such relations are extracted. This assures that needed metaclasses all can be extracted and extracted metaclasses all are right. It should be pointed out that when the algorithms extract metaclasses with OCL expressions, there may be superfluous metaclasses because OCL expressions are not deeply parsed in semantics.

Similarly, for the overall package structure of a metamodel, the algorithm 3 traverses packages according to the relations between them, i.e. inclusions, imports, and merges, and if and only if the packages between which there is such relations are extracted. This assures that needed packages all can be extracted and extracted packages all are right.

d) Tool Support

The algorithms used in the approach can be implemented in object-oriented modeling tools built-in metamodels without too much difficulty. The modeling tools can build package models and class models with package diagrams and class diagrams, respectively, and can save the models as XML files or the others. The built-in metamodels are also described in XML files or the others in the object-oriented modeling tools, and thus it is not difficult for the tools to show the built-in metamodels as package diagrams and meta-class diagrams. Therefore, it is feasible to integrate the module that implements the approach with the tools.

e) Compliance

According to explicit relations (such associations and inheritances) and implicit relations analyzed from OCL expressions, the approach only traverses and extracts elements from a metamodel for forming a new metamodel, and does not modify and add metamodel elements, and thus the new metamodel holds only necessary and sufficient metamodel elements according to the given pruning parameters. This means that all instances (models) of the pruned metamodel are also instances of the initial input metamodel, that is, the extracted metamodel still complies with the original one.

f) Limitation and Future Research

When handling OCL expressions in the approach, if a needed meta-class appears in an OCL expression, the other meta-classes in the expression all are related to the meta-class. In fact, some of the meta-classes are unrelated, and related meta-classes can be classified as direct correlative meta-classes and conditional correlative ones. If an OCL expression includes conditional statements, distinguishing different alternative segments of a metamodel (e.g. labeling or coloring) is better treatment. The above work can be finished by aid of an OCL parser.

As for the input of the pruning metamodel algorithms, what the paper gives are a metamodel, specified packages, and options, etc. We plan to consider the necessity to take meta-relations as the input of the pruning algorithms.

Our algorithms only can prune the metamodels like UML, and we also plan to study other kinds of metamodels to provide a general pruning approach.

g) Threats to Validity

Because we measure UML 2.4.1 in terms of counts for metamodel elements and relations between the elements, there is no threat to the measurements.

The metamodels like UML all are defined in MOF or extended based on UML, CWM, SysML, and SPEM etc. that also are defined in MOF, and thus the principle of building these metamodels is the same as UML's. Therefore, UML 2.4.1 is a representative of metamodels like UML in the paper and our algorithms have universality for pruning metamodels like UML.

A possible threat is lack of a formal proof of the correctness of the algorithms since the proof is a supplemental proof.

4 RELATED WORK

Sagar Sen et al., present a metamodel pruning algorithm (2009). They omit that package is an important mechanism for organizing the elements of a lager metamodel into groups, and think that the multiplicity * of UML is optional, and thus delete all meta-associations with *. In fact, multiplicity is a specification of the range of allowable cardinalities which an entity (including a relation) may assume (OMG, 2011a), and thus multiplicity * has specific semantics for defining metamodel elements. Their algorithm only considers the meta-classes appearing

in OCL expressions, and not the meta-classes related to these meta-classes just because the related meta-classes do not appear in the OCL expressions. Arnor Solberg et al. point out the importance of pruning metamodels from the aspects of model-driven development and aspect-oriented modeling (2009), but do not further give solutions. Jung Ho Bae et al., propose an algorithm for pruning small metamodels for seven types of UML diagrams (2008), and their algorithm does not consider packages and OCL expressions in UML metamodel and optional parameters.

For the comprehension and maintenance of metamodels, Strüber et al., present a tool that supports the decomposition of a meta-model into clusters of metamodel elements (Daniel et al., 2013). They apply clustering algorithms to obtain segments of metamodels, and our algorithm is for extracting the needed submetamodels that are complete in syntax and semantics according to the definition relations between metaclasses.

The static slicing technologies of class models are similar to ours. Jaiprakash et al. present an algorithm for static slicing of UML architecture models (2009), and their slicing criterion only consists of one class and one message. Huzefa Kagdi et al., propose an idea to enrich slicing criterion (2005), and only defines several concepts for context-free slicing of single UML class model. Fangjun et al propose a slicing algorithm for class diagrams (2004), and their algorithm is designed for dependence analysis for class diagrams by simply finding all relevant classes for a given class. Arnaud et al. present a language to build model slicers (2011), which can extract model slices from domain-specific models, and the built slicers can take dependent path length, optional classes, and optional properties as input, but do not take into account OCL expressions and packages, and thus their work is unsuited to prune metamodels.

5 CONCLUSIONS

The metamodels are important information sources with their own characteristics, and one only needs parts of the large and complex metamodels in many cases. According to the characteristics, the paper presents an approach to automatically bi-directionally extract needed part from a metamodel like UML by parsing network structure of packages and calculating metaclass models. Moreover, a pruned metamodel complies with its initial metamodel, and its size is agilely controlled with input options. The approach can service to a variety of applications that need to prune metamodels.

ACKNOWLEDGEMENTS

The work supported by the National Natural Science Foundation of China (No. 61672046).

REFERENCES

Arnaud Blouin et al. (2011) 'Modeling Model Slicers', *MoDELS, LNCS* 6981, 62-76.

Arnor Solberg, Robert France, and Raghu Reddy. (2005) 'Navigating the MetaMuddle'. *In: Proceedings of the 4th Workshop in Software Engineering and Application*, Jamaica, 2005, 315-321.

Brian Henderson-Sellers. (2005) 'UML– the Good, the Bad or the Ugly? Perspectives from a panel of experts', *Software System Model*, 2005(4), 4–13.

Daniel Strüber,Matthias Selter, and Gabriele Taentzer. (2013) 'A Tool support for clustering large meta-models'. Proceeding. *In: Proceedings of the Workshop on scalability in model driven engineering*, NY, USA ,2013. ACM New York, 1-4.

Dori Dov. (2002) 'Why significant UML change is unlikely', *Communications of the ACM*, 45(11), 82–85.

Fangjun W. and Tong Y. (2004) 'Dependence analysis for UML class diagrams', *Journal of Electronics*, 2004, 21(3), 249–254.

Flatscher RG. (2002) 'Metamodeling in EIA/CDIF – Meta-metamodel and Metamodels' *ACM Trans. Modeling and Computer Simulation*, 12(4), 322–342.

Frédéric Fondement, Pierre-Alain Muller, Laurent Thiry, Brice Wittmann, and Germain Forestier. (2013) 'Big Metamodels Are Evil'. *Model-Driven Engineering Languages and Systems, Lecture Notes in Computer Science*, Volume 8107, 2013,138-153.

Huzefa Kagdi, Jonathan I. Maletic, Andrew Sutton. (2005) 'Context-Free Slicing of UML Class Models'. *In: Proceedings of the IEEE International Conference on Software Maintenance. Washington*, 2005,635 – 638.

Jaiprakash T. Lallchandani and R.Mall. (2009) 'Static Slicing of UML Architectural Models', *Journal of Object Technology*, Vol. 8(1), 159-188.

Jiang Yangbing, Weizhong Shao, Lu Zhang, Zhiyi Ma, Haohai Ma. (2004) 'On the Classification of UML's Meta Model Extension Mechanism', *Lecture Notes in Computer Science*,3273, 54-68.

Jung Ho Bae, Heung Seok Chae. (2008). 'UMLSlicer: A Tool for Modularizing the UML Metamodel using Slicing'. *In: Proceedings of the IEEE 8th International Conference on Computer and Information Technology*, Sydney, 2008,772–777.

Ma Zhiyi, Xiao He, Chao Liu. (2013) 'Assessing the quality of metamodels', *Frontiers of Computer*

Science,Volume 7(4), 558-570.
OMG. (2003) UML 2.0 OCL Specification. OMG ptc/03-10-14.
OMG.(2011a) Unified Modeling Language Superstructure Version 2.4.1. OMG formal/2011-08-06.
OMG. (2011b) Unified Modeling Language Infrastructure Version 2.4.1. OMG formal/2011-08-05.
OMG. (2011c) Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. OMG formal/2011-01-01.
OMG. (2013) Meta Object Facility (MOF) 2.4.1. OMG formal/2013-06-01.
Robert France and Bernhard Rumpe. (2007) 'Model-driven Development of Complex Software: A Research Roadmap'. *In: Proceedings of the 2007 Future of Software Engineering, France*.IEEE,37–54.
Sagar Sen, Naouel Moha, Benoit Baudry, and Jean-Marc jezequel. (2009) 'Metamodel Pruning', *MoDELS, LNCS* 5795, 32-46.