HAL
open science

Modeling Model Slicers

Arnaud Blouin, Benoit Combemale, Benoit Baudry, Olivier Beaudoux

To cite this version:

Arnaud Blouin, Benoit Combemale, Benoit Baudry, Olivier Beaudoux. Modeling Model Slicers. ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems, Oct 2011, Wellington, New Zealand. pp.62–76, 10.1007/978-3-642-24485-8_6 . inria-00609072v2

HAL Id: inria-00609072

https://inria.hal.science/inria-00609072v2

Submitted on 25 Feb 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Modeling Model Slicers*

Arnaud Blouin[1], Benoît Combemale[1], Benoit Baudry[1], Olivier Beaudoux[2]

[1] IRISA/INRIA, Triskell Team, Rennes, France
{ablouin,bcombemale}@irisa.fr benoit.baudry@inria.fr
[2] GRI-ESEO, Angers, France
olivier.beaudoux@eseo.fr

Abstract. Among model comprehension tools, model slicers are tools that extract a subset from a model, for a specific purpose. Model slicers are tools that let modelers rapidly gather relevant knowledge from large models. However, existing slicers are dedicated to one modeling language. This is an issue when we observe that new domain specific modeling languages (DSMLs), for which we want slicing abilities, are created almost on a daily basis. This paper proposes the Kompren language to model and generate model slicers for any DSL (*e.g.* software development and building architecture) and for different purposes (*e.g.* monitoring and model comprehension). Kompren's abilities for model slicers construction is based on case studies from various domains.

An extended version of this paper has been published in the SoSyM Journal, see [4] and `http://hal.archives-ouvertes.fr/hal-00746566/`.

1 Introduction

Model slicing is a model comprehension technique inspired by program slicing [17]. This consists in extracting a subset of a model, called a *slice*. A slice has different forms depending on its purpose. For example, when trying to understand a large class diagram, it can help to extract the smallest strongly connected graph that is the subset of the class diagram that represents all dependencies of a particular class of interest. On the other hand for another comprehension purpose, one might want a slice that is closer to what a semantic zoom could provide [5], *e.g.* provide a flat view of all references and attributes inherited by a class of interest.

There has been previous work on the definition of model slicers. For example, [11] proposed model slicers for UML class and state diagrams. However, all existing model slicers are dedicated to extracting one form of slice from models that conform to a specific metamodel. In times when new domain specific modeling languages (DSMLs) appear regularly to improve productivity and increase the adoption of model-driven engineering, this becomes an issue: on one hand it is not convenient to develop slicers from scratch for every new DSML; on the other hand these DSMLs will provide full expected benefits for productivity only if they are supported by the same analysis and

* This work is partially supported by the EU FP7-ICT-2009.1.4 Project N°256980, NESSoS: Network of Excellence on Engineering Secure Future Internet Software Services and Systems.

comprehension tools as general purpose languages. Thus, it is necessary to develop a generative approach that will automatically build model slicers for new metamodels.

In this paper we propose Kompren[3], a DSML to model model slicers for a particular domain (captured in a metamodel). We learn from existing model slicers, as well as from practical experiences that require the extraction of sub parts out of models. This learning phase leads to the different features of the Kompren language. Kompren mainly allows the selection of classes and properties in an input metamodel. By default, the model slicer generated out of these elements will be such that it builds slices that contain all instances of the selected classes and properties, plus all necessary elements to make the slice a valid instance of the input metamodel. Kompren also offers a set of language features to generate model slicers that can still be parameterized in order to process the model slice for a specific purpose. These different characteristics of Kompren aim at achieving two goals for our generative approach: automatically build model slicers for any DSML; have model slicers that can extract different forms of slices, depending on the purpose of the slice.

The contributions of this paper are the following:

- a language to model model slicers for any metamodel
- a compiler that automatically generates model slicers
- demonstrations of the language expressiveness over three illustrative cases.

In section 2 we introduce several motivating scenarios that illustrate the various forms of model slices that must be generated when analyzing models in various languages. Section 3 introduces the overview of building model slicers with the Kompren language. Section 4 presents the Kompren language: its metamodel, compiler and concrete syntax. Section 5 demonstrates the expressiveness of Kompren on three illustrative cases. Section 6 discusses related work and section 7 concludes this work.

2 Heterogeneous Use Cases of Model Slicing

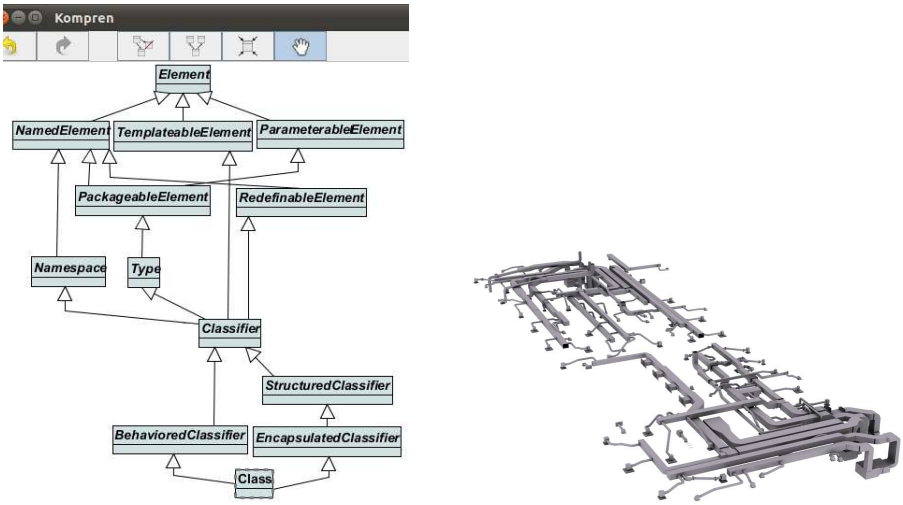
The classical use of model slicing consists in extracting sub-models from models by keeping conformance rules. However, as shown in the motivating use cases below model comprehension also requires extracting models which do not satisfy conformance. Still, this extraction can rely on model slicing mechanism.

Use case 1: Model operation analysis. Given a model operation on a large metamodel MM_1, developers want to get the *effective* metamodel MM_2 used by the operation such that $MM_2 \subset MM_1$. For instance, when defining a state machine flattening operation over the UML metamodel, only the UML class diagram and the UML state machine elements are used. This model operation must be analyzed to select MM_1 elements it uses and to get the effective metamodel MM_2 [13].

Use case 2: Semantic zooming on models. Understanding and manipulating large models require visualization techniques to provide meaningful navigation capabilities

[3] https://www.irisa.fr/triskell/Softwares/protos/kompren/

[16]. Semantic zooming is a Human-Computer Interaction (HCI) that can be applied for this purpose. In contrast to physical zooming that changes the size of objects, semantic zooming changes the type and meaning of information displayed by objects [5]. For instance, as shown in Fig. 1a, semantically zooming on class inheritance extracts super-classes of a given class. We can notice that semantic zooming is different from model slicing for two reasons: the extracted slice does not necessarily conform to the metamodel and is not saved as a new model but used by HCI to perform semantic zooming.



(a) Viewing Super-classes of the UML Class *Class*

(b) Complex Mechanical Model of a Building, extracted from [15]

Fig. 1. Examples of Semantic Zooms

Model slicing and semantic zooming are not limited to the computer science domain. In the design and construction industry, recent works proposed a model-driven approach for the interoperability of building models [15]. Such models are complex and need tools to extract information relevant to a particular concern and stakeholder. For example, Fig. 1b shows the mechanical model of a building. Mechanical model stakeholders may want to focus on the details of a given location or mechanism of the building.

Use case 3: Model Monitoring at runtime. Monitoring models at runtime is an important feature to control their evolution. For example, component-based model stakeholders may want to monitor only component activations among all the different possible modifications. Thus, dedicated tools need to extract only information relevant to component activation. Such information must be incrementally extracted to improve performance on large models.

3 Overview

Figure 2 provides an overview of the proposed approach to model model slicers. The core contribution of this paper is a modeling language dedicated to the construction of model slicers. The language is called *Kompren*. All the concepts and relations of *Kompren* are captured in a model slicer metamodel (MSMM at the top of figure 2). A model slicer model (MSM) expressed with *Kompren* refers to a set of classes and relations from the input *metamodel*. Instances of the referenced classes and relations will be selected for slicing in the input model. Consequently, MSMM points to elements Ecore to enable *Kompren* models to use elements from an *input metamodel*. MSMM also points to Kermeta, an action language used to specify the behavior of a slicer. *Kompren*'s compiler processes a *Kompren* model defined for an *input metamodel*, and automatically generates an actual *model slicer function* (MSF).

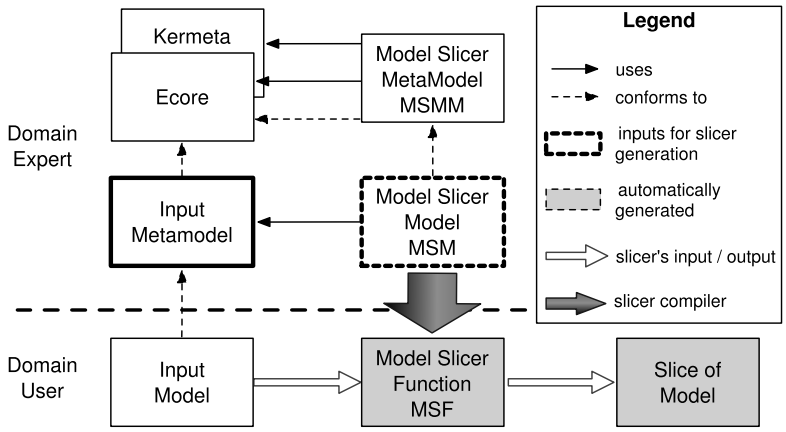


Fig. 2. Overview for Modeling Model Slicers with Kompren

The *Kompren* model can defined elements that are generated as parameters for the *model slicer function*. These parameters allow adjusting the slicing process to an actual instance of the *input metamodel*. Once the function's parameters are set, the *model slicer function* processes an *input model* to automatically extract a model slice from it.

This global approach is a two-level generation process: *Kompren*'s compiler generates a *model slicer function*, which in turn generates a model slice. From a methodological perspective, we also distinguish two roles for *Kompren* users:

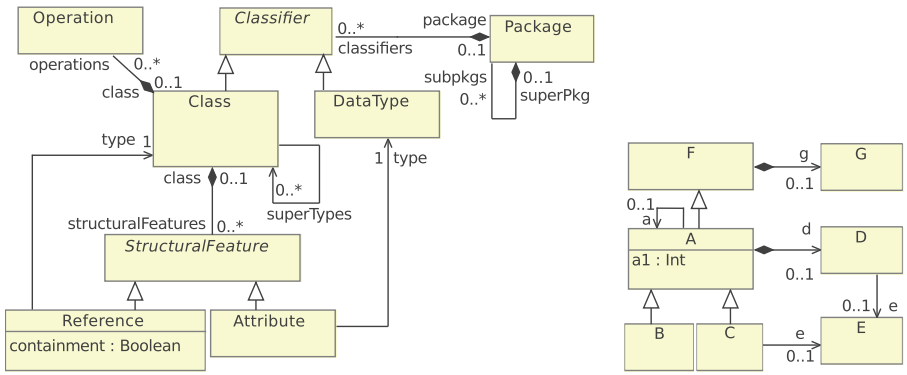
- **Domain expert**. The domain expert knows the domain captured in the *input metamodel* and knows its concepts and relationships. This person is thus in charge of leveraging this domain in order to model one or several model slicers that are relevant for this domain. The *domain expert selects the elements in the metamodel* that will be processed by the model slicer.
- **Domain users** create models in the domain. These users, through their modeling activities can create large instances of the *input metamodel*. At some point they

need to extract slices thanks to the *model slicer function*. These *users parameterize the model slicer according to their need and according to the values in the instance*.

4 Model-Driven Specification of Slicers

4.1 Expected Features for a Model Slicer

Basically a *Model Slicer Model* (MSM) enables the specification of classes and properties whose instances must be selected from a given *input model*. Input models can be either structural or behavioral. In both cases, their slicing consists in slicing the structure of their metamodel. We distinguish two generation modes of a *model slicing function* (MSF) from a MSM. Below, we detail and illustrate these two modes through examples based on the class diagram *input metamodel* (Fig. 3a) and the *input model* shown in Fig. 3b.

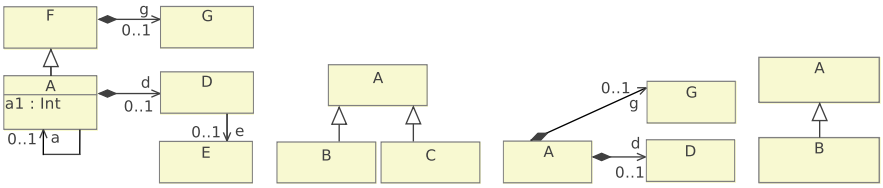


(a) Input Class Metamodel **(b)** Input Class Model

Fig. 3. Class Model Example

- The *strict* mode (by default) generates a MSF that extracts model slices that satisfy all the structural constraints imposed by the input metamodel. Thus, by default a slice is a valid instance of the *input metamodel*.
  For example, Fig. 4a is a strict slice of Fig. 3b that conforms to the class diagram *input metamodel*.
- The *soft* mode relaxes the conformity constraint over model slices (ensured by the strict mode) in exchange of additional features for model slicer modeling. This mode is an answer to the usages illustrated in the motivating examples where slices are not instances of the *input metamodel*. In particular, the previous examples have motivated the need for the following features in the soft mode:
  - **Add an opposite property in the input metamodel.** For example, Fig. 4b is a slice of 3b that selects `A` and its subclasses. To ease the slicing of the *input model*, the MSM requires the opposite of the `superTypes` property in the *input metamodel*.

- **Add constraints to filter the sliced elements.** For example, Fig. 4c is a slice of 3b that selects `A` and only its composite references. Similarly, Fig. 4d is a slice of 3b that selects `B` and its supertypes within a radius of 1.
- **Enlarge the slicing output format.** For example, instead of saving the sliced elements, they could be used to print their relative information. Other usages such as the notification of external tools must be also considered.
- **Automatically update slices.** On input model changes, the MSF automatically updates the slice.



(a) Strict Mode **(b)** Opposite **(c)** Constraint **(d)** Radius

Fig. 4. Class Model Slices

4.2 Kompren Abstract Syntax

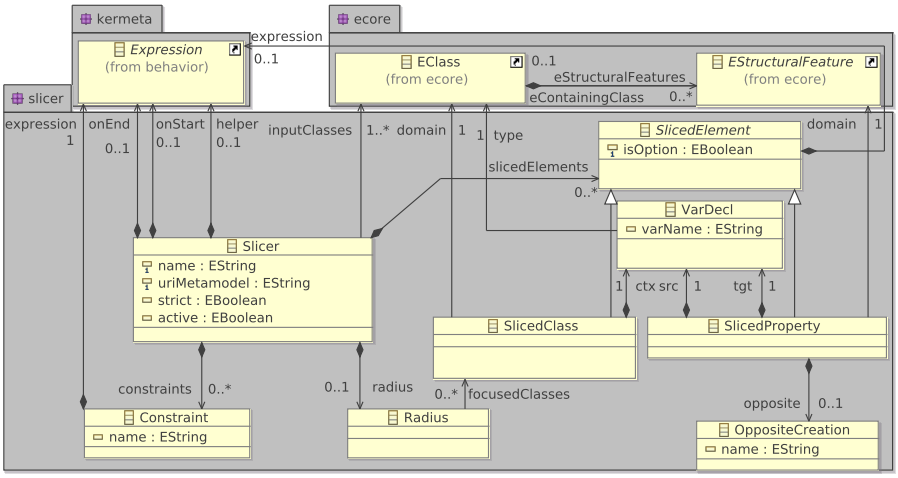


Fig. 5. Model Slicer metamodel

The metamodel shown in Fig. 5 describes the abstract syntax of Kompren. An instance of this metamodel is a *Model Slicer Model* (MSM). The main package is *slicer*.

In this package, a *Slicer* is mainly composed of *SlicedElements*. These sliced elements correspond to the classes (*SlicedClass*) and the properties (*SlicedProperty*) of interest in the *Model Slicing Function* (MSF). All sliced elements belong to the *input metamodel* identified in the slicer by its URI (*uriMetamodel*). Optional *SlicedElements* (*i.e. isOption* is true) are options of the generated MSF. This lets the domain user choose whether an element is selected or not.

A *SlicedClass* refers to a class (*EClass*) in the *input metamodel* (*domain*). All instances of a referenced class in a given *input model* are selected by the MSF. Then *ctx* (contained in *SlicedClass*) serves as a temporary variable to successively manipulate each instance (*i.e.* an iterator). The type of this iterator (*type* in *VarDecl*) must correspond to the sliced class. This constraint can be formalized using OCL as follows:

```
1 context SlicedClass inv:
2   self.domain = self.ctx.type
```

Similarly, a *SlicedProperty* refers to a property (*EStructuralFeature*) in the *input metamodel*. All instances of a referenced property in an *input model* are selected by the MSF. The *src* and *tgt* iterators allow the manipulation of the property's source and target. The types of these iterators correspond to the source and the target class of the property:

```
1 context SlicedProperty inv:
2   (self.domain.eContainingClass = self.src.type) &&
3   (self.domain.eType = self.tgt.type)
```

In addition, a sliced property may define an *OppositeCreation* to precise the creation of an opposite property whose the role is given by the *name*.

We assume in this paper an *input metamodel* defined with an existing object-oriented metamodeling language. We use in our experiments the Ecore metamodeling language provided by the *Eclipse Modeling Framework*[4] whose elements are imported in the package *ecore*. In Ecore, a class and a property are identified by respectively an *EClass* and an *EStructuralFeature*. Another object-oriented metamodeling language could be easily considered in Kompren.

Moreover, the iterators on sliced elements (instances of the specified *SlicedClass* and *SlicedProperty*) allow the domain expert to express the expected behavior for each selected instance. The effect of the MSF on each selected instance is described as an expression using an action language. In our experiments, we use the action language of Kermeta [12] whose the corresponding metamodel is imported in the package *kermeta*. Another action language could be easily considered in Kompren.

The two modes previously introduced in Section 3 are supported by Kompren. By default, a MSF is generated according to the *strict* mode. By setting the attribute *strict* (in *Slicer*) to false, the MSF is generated according to the *soft* mode.

In that case, the remaining concepts in the Kompren metamodel are used to specify specific behaviors of the generated MSF. The expressions *onStart* and *onEnd* are used to add a particular behavior in the MSF, which are respectively applied before and after the visit of the *input model*. Expressions defined to bring executability to slicers may require classes provided by third party libraries, attributes or operations needed to the

[4] http://www.eclipse.org/modeling/

slicing process. Thus, the domain expert can specify an *helper* that will contain this information.

The *radius* and the *constraints* can be used to filter the sliced element in the *input model*. The *radius* precises in the MSM the *focusedClasses* for which the MSF should be limited to a selection within a given radius. The focused classes must be included in the sliced classes that can be formalized as follows:

```
1 context Slicer inv:
2   not self.radius.oclIsUndefined() implies
3   self.slicedElements->select{c | c.isTypeOf(SlicedClass)
4   }->includeAll(self.radius.focusedClasses)
```

The value of the radius must be specified by the domain user as a parameter of the MSF. The *constraints* allow the domain expert to define a condition that must be respected to trigger the slicing of the element targeted by the condition.

The *inputClasses* precise the type of instances that the MSF will take as input to start the slicing.

Finally, the attribute *active* permits to specify if the MSF must be executed as a batch or an active process. By default, the generated MSF is a batch process executed a single time on the input model. By settings the attribute *active* to true, the generated MSF is executed a first time and then observes modifications applied on the input model in order to incrementally update the slice.

4.3 Concrete Syntax

A textual concrete syntax has been defined for Kompren allowing the domain expert to define a *Model Slicer Model* (MSM). As an example, the following listing shows the *active* and *soft* MSM *ClassModelSlicer* (*cf.* line 1), for the metamodel in Fig. 3a (*cf.* line 2). The classes of the instances used to launch the *Model Slicing Function* (MSF) are declared line 3.

Thereafter, line 4 specifies a sliced class while lines 5 to 8 specify sliced properties. An expression defined for the sliced class `Class` is described line 4 where *cl* refers to the context of the sliced class. An optional property is illustrated line 5 thanks to the keyword `option`. An opposite to a property is defined thanks to the keyword `opposite` as shown line 6 where *lowerTypes* is the name of the opposite.

Line 9 illustrates how to declare a radius based on `Class` to limit the selection in the *input model* by the MSF. The definition of a constraint consists in specifying a Kermeta boolean expression as shown line 10. Lines 11 to 13 illustrate the definition of the preprocessing, the post-processing and the helper of the slicer.

```
1  slicer active soft ClassModelSlicer {
2   domain: platform:/resource/classModel.ecore
3   input: Class
4   slicedClass: Class cl{ stdio.writeln(cl.name) }
5   slicedProperty: Class.superTypes option
6   slicedProperty: Class.superTypes opposite(lowerTypes)
7   slicedProperty: Class.structuralFeatures
8   slicedProperty: Reference.type
9   radius: Class
10  constraint: Reference.containment
11  onStart { stdio.writeln("Starting slicing") }
```

```
12  onEnd   { stdio.writeln("Ending slicing") }
13  helper  { /* Definition of the helper */ }
14 }
```

4.4 Semantic

As defined in Fig. 2, model slicer models (MSM) are compiled into model slicer functions (MSF). This compilation produces Kermeta programs composed of two parts. The first part augments the input metamodel with required information. These information are the opposites specified in MSMs and methods used to explore the input model. These methods are generated for the metamodel elements selected in MSMs. If the slicer is defined as strict, these methods are also generated for elements not selected in MSMs but required to assure the semantic properties.

The second part generates the slicer function. The preprocessing (*onStart*) and the post-processing (*onEnd*) methods and the Kermeta code corresponding to the helper are created. From the input classes, the radius and the constraints defined in MSMs are generated as parameters of the slicer function. For instance, the following Kermeta code illustrates such generation where: *launch* is the operation that starts the slicing; *inputClass:Class[0..*]* defines the *Class* instances used to launch the slicing; *radius:Integer* specifies the slicing radius; *composition:Boolean* is a constraint that declares if only composition references must be sliced.

```
operation launch(inputClass:Class[0..*], radius:Integer,
                 composition:Boolean)
```

Once generated, the slicer function can be executed by calling the launch operation with its required parameters. The preprocessing is first executed. Then begins the exploration of the input model using the input instances given as parameter. Each of these instances is visited. Visiting an instance or a property consists in executing the associated behavior: for strict slicers, adding the sliced instance to a new model; for soft slicers, executing the corresponding Kermeta expression defined by the developer. Each selected property of the current visited class instance are then explored (if they satisfy the constraints defined in MSMs) to recursively explore their target class instance.

Starting at 0, a value is incremented on each visited class instance concerned by the radius. The slicing process thus stops when no elements can be sliced anymore or when this value is greater than the radius given as parameter. When the slicing has stopped, the post-processing is executed.

About *active* slicers, because Kermeta does not manage observability of Ecore models, we use the *ActiveKermeta* toolkit [3]. ActiveKermeta replaces Kermeta batch operations, such as *c.each{e|...}* that visits each element *e* of collection *c*, by active operations, such as *c.eachAdded{e | ...}* supplemented by *c.eachRemoved{e | ...}* that are respectively called when *e* is added or removed from *c*.

5 Validation

In this section, we apply our model slicing approach to three heterogeneous case studies illustrating the main usages that can be done using our approach.

5.1 Model Operation Analysis

Extracting static metamodel footprint for a model operation defined over a metamodel MM_1 (in our case the Kermeta metamodel) consists in extracting the elements of MM_1 used by the operations [6]. In this section, we use Kompren to model the footprint generator proposed by Jeanneret *et al.* [6] and the metamodel pruner proposed by Sen *et al.* [13]. The Kompren model is smaller than the initial model slicers: around 70 LoC have been needed (see details below) while the static metamodel footprinting and the metamodel pruner both required around 1200 Kermeta LoC. This use case illustrates the ability of Kompren to ease the slicer definition process.

The effective metamodel extraction is performed through two model slicers: a first slicer analyzes the model operation to extract the metamodel footprint, *i.e.* the list of MM_1 elements used by the operation; a second slicer uses this footprint to extract the effective metamodel from MM_1. The effective metamodel extraction could have been defined using a single slicer. We divided this operation into two slicers to separate the concerns and be modular.

The first slicer extracts the list of MM_1 elements used by the operation. Since this slice does not conform to MM_1, we model the slicer in *soft* mode (line 1). The model operation is implemented in Kermeta. Thus, it is an instance of the Kermeta metamodel MM_{op} and the slicer explores classes and properties of MM_{op} (lines 5 to 15). The result of the slicing function will be the list of classes used in the operation (line 4). This list is defined in the helper (line 17). By default all the classes, that can come from either MM_1or MM_{op}, are explored. Because only the classes from MM_1 must be stored, a helper is defined to select them (lines 18 to 22).

```
1  slicer soft OperationStaticAnalysis {
2   domain: platform:/resource/kermeta.language.model/src/main/ecore/kermeta.ecore
3   input : kermeta.structure.ModelingUnit // The model operation to analyse.
4   slicedClass: kermeta.structure.ClassDefinition cd { addClassDefinition(cd) }
5   slicedProperty: kermeta.structure.ModelingUnit.packages
6   slicedProperty: kermeta.structure.Package.ownedTypeDefinition
7   slicedProperty: kermeta.structure.ClassDefinition.ownedOperation
8   slicedProperty: kermeta.structure.ClassDefinition.ownedAttribute
9   slicedProperty: kermeta.structure.Operation.ownedParameter
10  slicedProperty: kermeta.structure.TypedElement.type
11  slicedProperty: kermeta.structure.ParameterizedType.typeDefinition
12  slicedProperty: kermeta.structure.Operation.body
13  slicedProperty: kermeta.behavior.VariableDecl.type
14  slicedProperty: kermeta.behavior.Block.statement
15  //... 29 properties of MMop are sliced.
16  helper {
17   reference metamodelClassesUsed : ClassDefinition[0..*]
18   reference inputMetamodel : ModelingUnit
19   //... Load of the input metamodel.
20   operation addClassDefinition(cd : ClassDefinition) : Void is do
21    if(inputMetamodel.contains(cd)) then metamodelClassesUsed.add(cd) end
22   end
23 }}
```

The second slicer, modeled as follows, uses the footprint computed by the first one. This slicer is modeled in *strict* mode (line 1) to create an output model that is a strict slice of the input metamodel MM_1 (specified line 2). This slicer slices all the classes (line 4) linked to the input classes by inheritance or properties (lines 10 to 12). All properties and operations of the class sliced are included (lines 5 to 9). Because *ClassDefinition* is linked to *Package* by a 1..1 reference, this relation and its target class must

be sliced to extract a strict slice. Since we model in strict mode, the packages containing sliced elements are sliced even if *Package* is not modeled as a *slicedClass*. This mode also includes $1..n$ attributes of classes *ClassDefinition*, *Property* and *Operation*.

```
slicer strict MetamodelFootprintExtraction {
 domain: platform:/resource/kermeta.language.model/src/main/ecore/kermeta.ecore
 input : kermeta.structure.ClassDefinition
 slicedClass: kermeta.structure.ClassDefinition
 slicedClass: kermeta.structure.Property
 slicedClass: kermeta.structure.Operation
 slicedProperty: kermeta.structure.ClassDefinition.ownedAttribute
 slicedProperty: kermeta.structure.ClassDefinition.ownedOperation
 slicedProperty: kermeta.structure.Operation.ownedParameter
 slicedProperty: kermeta.structure.TypedElement.type
 slicedProperty: kermeta.structure.TypedDefinition.superType
 slicedProperty: kermeta.structure.ParameterizedType.typeDefinition
}
```

5.2 Bringing Semantic Zoom to Model Visualization

Model slicing can be used to bring semantic zooming to model visualization. In this case, the slicer defines which classes and relations of the visualized model must be displayed in the user interface (UI). For example, the following code defines a slicer that slices Kermeta models. Because the goal of this slicer is to notify the UI about sliced elements, it is defined as *soft* (line 1). It takes as input instances of *ClassDefinition* (line 3) selected by users using the UI. As shown in Fig. 6, the UI displays classes, inheritances and properties. At the beginning of the slicing all these model elements are hidden (line 6). Then, when model elements are sliced, the UI is notified that they must be shown (lines 9, 11 and 14). At the end of the slicing, the UI is updated to perform the graphical changes (line 7). Some properties must be explored to access the instances to slice (lines 13 to 17). All these properties to slice are defined as optional. Thus, for each feature of the model visualizer (*e.g.* showing the inheritance tree of a selected class), developers can define which properties must be explored.

```
1  slicer soft kermetaSemanticZoom {
2   domain: platform:/resource/kermeta.language.model/src/main/ecore/kermeta.ecore
3   input: kermeta.language.structure.ClassDefinition
4   radius: kermeta.language.structure.ClassDefinition
5   constraint: kermeta.language.structure.Property.lower>0
6   onStart { extern ClassDiagramView.hideAllElements() }
7   onEnd   { extern ClassDiagramView.updateView()      }
8   slicedClass: kermeta.structure.ClassDefinition cd{
9    extern EntityView.showClass(cd) }
10  slicedClass: kermeta.structure.Property prop {
11   extern ReferenceView.showReference(prop.name, prop.owningClass,
12                                      prop.type.asType(Class).typeDefinition) }
13  slicedProperty: kermeta.structure.TypeDefinition.superType option src tar{
14   extern InheritanceView.showInheritance(src, tar.asType(Class).typeDefinition) }
15  slicedProperty: kermeta.structure.ParameterizedType.typeDefinition option
16  slicedProperty: kermeta.structure.ClassDefinition.ownedAttribute option
17  slicedProperty: kermeta.structure.TypedElement.type option
18 }
```

The UI shown in Fig. 6 provides a spinner that permits to define the radius effect of the slicing (defined line 4). The UI also provides a check-box called ”With card 0”. This check-box permits to set if properties which lower cardinality equals 0 must be sliced

or not (line 5). The graphical representation of the model and the widgets of the UI are defined separately from the slicer.

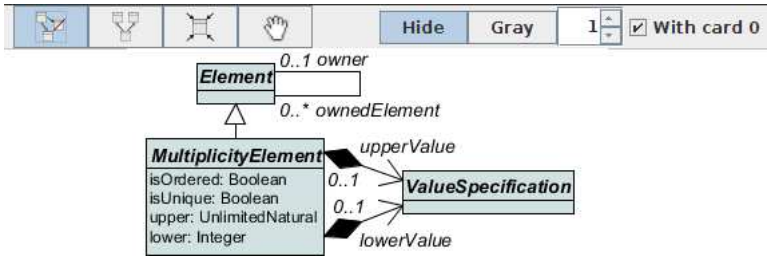
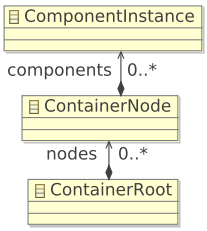


Fig. 6. Class Diagram Visualizer Providing Semantic Zooming Features

5.3 Monitoring Component-based Models at Runtime

Our model slicing approach can also be used to slice models at runtime, *i.e.* the slicing process is no more a batch process but is sustained at runtime to re-evaluate model elements that change. For example, Kevoree is a component-based model that manages addition and removal of components at runtime[5]. These changes can be monitored to provide stakeholders with such information.



(a) Excerpt of the Kevoree Metamodel

```
1  slicer active soft KevoreeComponentMonitoring {
2   domain: platform:/resource/kevoree/kevoree.ecore
3   input : kevoree.ContainerRoot
4   slicedClass: kevoree.ContainerNode
5   slicedClass: kevoree.ComponentInstance ci
6    { table.addComponent(ci) }
7    { table.removeComponent(ci) }
8   slicedProperty: kevoree.ContainerRoot.nodes
9   slicedProperty: kevoree.ContainerNode.components
10  helper {
11   require "platform:/resource/kermeta/ComponentTable.km"
12   attribute table: ComponentTable
13 }}
```

(b) The slicer model

Fig. 7. Model Slicer Model for Monitoring Kevoree Component Additions and Removals

Fig. 7a is the excerpt of the Kevoree metamodel related to component additions and removals. Fig. 7b gives the model slicer model (MSM) dedicated to the slicing at

[5] http://dist.kevoree.org/

runtime of component additions and removals. In the Kevoree metamodel, activated components are contained into the composition *components* of class *ContainerNode*. The component model can contains several node containers (composition *nodes*). Thus, these two compositions *components* and *nodes* are selected by the slicer (lines 8 and 9). Classes *ContainerNode* and *ComponentInstance* are defined as the classes to slice (lines 4 and 5). The input instances given to the active slicer are *ContainerRoot* instances (line 3). This MSM differs from the previous "batch" slicers in two points. Firstly line 1, the keyword *active* means that the generated slicer function must remain active by updating the sliced output model whenever the input model changes. Secondly, the attribute *table* (defined in the helper line 12) is managed throughout two subsequent blocks: similarly to batch slicers the first block defines how to update the table whenever a new component instance *ci* appears (line 6); the second block defines how to update the table whenever *ci* is removed (line 7).

6 Related Work

Although model slicing has been studied in literature, most of the inventoried approaches focus on a particular DSML. For instance, [7,2,11,10,14] focus on the slicing of UML models whereas [9] proposes the slicing of state-based models. Because of the diversity of DSMLs, our approach aims at being more generic to allow the specification of slicers for any DSML. Our generative approach aims at reducing the programmatic effort spent for the development of model slicers, while giving domain users the ability to customize the application of the MSF (*e.g.* radius).

We identified two kinds of output produced by the slicers of the current approaches. In the first case, the output is a model that conforms to the input metamodel, such as in [13,11,8]. In the second case, the output is a model that may be not conform to the input metamodel, such as in [6]. A key concern that our slicing proposal insists on is the ability for developers to define the kind of output they want. For example, a strict slicer will produce models that conform the input metamodel with respect to the model slicing definition. But we also identified several use cases, such as semantic zooming or model operation analysis, where the expected output is neither a model that conforms to the input metamodel nor even a model. Thus our slicing proposal permits developers to define soft slicers which output is customizable.

Androutsopoulos *et al.* [1] propose different finite state machine slicing algorithms. Their basic slicer removes a set of transitions to ignore and useless states from finite state machines. This algorithm can be performed using our approach by defining parameters that state the slicer not to slice transitions having given names. Their other algorithms extend the first one by removing untriggerable transitions and merging states having identical semantics. Our approach does not permit to define such slicers.

Kelsen *et al.* [8] propose an approach for decomposing models into sub-models to tame the complexity of large models. This approach has similarities with ours since they are both not dedicated to a unique DSML and they can extract sub-models of interest that still conforms to the input metamodel. However, their approach does not permit developers to specify the slicing process, *i.e.* to select which elements of the input models must be sliced, and is restricted to the strict model slicing usage.

Shaikh *et al.* [14] use model slicing for verification purpose. The goal of this approach is to check if an input UML model supplemented by OCL constraints has legal instances. OCL constraints are thus analyzed and interpreted to identify which model elements are constrained. If their application is dedicated to one of our use case (model operation analysis), such OCL analyzes and interpretation is much more complex than extracting types.

Lallchandani *et al.* [10] propose a slicing technique for UML architectural models. Even if the proposed approach is limited to UML architectural models, it uses slicing for different purposes such as regression testing and understanding large architectures.

Obeo Designer[6] offers the possibility to easily create graphical viewpoints on large models. The representation of a slice can be seen as a viewpoint. However, the tool is limited to visualization and does not address manipulation or serialization of the slices.

7 Conclusion

A number of recent work inspired by program slicing [17] have proposed operations that extract sub parts of models for different purposes [14,11,8,6]. These operations are extremely helpful to assist comprehension when building large models. With the growing adoption of domain-specific modeling, these model comprehension abilities should be available for any domain-specific modeling language. However, all existing model slicing approaches are dedicated to one modeling language and one form of slice.

In this work we analyze needs for model slicing to precisely identify expected features for domain-specific model slicers. The major contribution of this paper is the *Kompren* language to model a model slicer for a domain-specific metamodel. We develop a two-level generative approach on the basis of Kompren: Kompren's compiler processes Kompren models to automatically generate an actual model slicer; this slicer can in turn automatically extract model slices from domain-specific models.

This paper presents the details of Kompren's features, abstract and concrete syntax and compiler. We also demonstrate Kompren's expressiveness through three different cases that aim at slicing three different forms of slices in three different domains. In particular we model the slicers defined by Jeanneret *et al.* [6] and by Sen *et al.* [13] and show that the Kompren models (*a.k.a.* model slicer models) are much smaller and easier to understand and evolve than the original slicers.

Following our evaluation on the expressiveness of our language, we plan to experiment the scalability of our approach. It could be interesting to explore MSM debugging as well.

References

1. Androutsopoulos, K., Binkley, D., Clark, D., Gold, N., Harman, M., Lano, K., Li, Z.: Model projection: Simplifying models in response to restricting the environment. In: International Conference on Software Engineering (ICSE'11) (2011)
2. Bae, J.H., Lee, K., Chae, H.S.: Modularization of the UML metamodel using model slicing. In: Proc. of the IEEE Inter. Conference on Information Technology. pp. 1253–1254 (2008)

[6] http://obeo.fr/pages/obeo-designer

3. Beaudoux, O., Blouin, A., Barais, O., Jézéquel, J.M.: Active operations on collections. In: MoDELS'10: Proc. of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. pp. 91–105 (2010)
4. Blouin, A., Combemale, B., Baudry, B., Beaudoux, O.: Kompren: Modeling and generating model slicers. Software and Systems Modeling (SoSyM) pp. 1–17 (2012)
5. Herman, I., Melançon, G., Marshall, M.S.: Graph visualization and navigation in information visualization: A survey. IEEE Trans on Visual Comput Graph 6, 24–43 (2000)
6. Jeanneret, C., Glinz, M., Baudry, B.: Estimating footprints of model operations. In: International Conference on Software Engineering (ICSE'11) (2011)
7. Kagdi, H., Maletic, J.I., Sutton, A.: Context-free slicing of uml class models. In: Proc. of the IEEE International Conference on Software Maintenance. pp. 635–638 (2005)
8. Kelsen, P., Ma, Q., Glodt, C.: Models within models: Taming model complexity using the sub-model lattice. In: In proc. of International Conference on Fundamental Approaches to Software Engineering, FASE'11. pp. 171–185 (2011)
9. Korel, B., Singh, I., Tahat, L., Vaysburg, B.: Slicing of state-based models. In: Proc. of the IEEE International Conference on Software Maintenance (ICSM'03) (2003)
10. Lallchandani, J.T., Mall, R.: A dynamic slicing technique for uml architectural models. IEEE Transactions on Software Engineering 99 (2010)
11. Lano, K., Kolahdouz-Rahimi, S.: Slicing of uml models using model transformations. In: International Conference on Model Driven Engineering Languages and Systems (MODELS'10), pp. 228–242 (2010)
12. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In: Proceedings of MODELS/UML'2005. pp. 264–278 (2005)
13. Sen, S., Moha, N., Baudry, B., Jézéquel, J.M.: Meta-model Pruning. In: 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09) (2009)
14. Shaikh, A., Clarisó, R., Wiil, U.K., Memon, N.: Verification-driven slicing of uml/ocl models. In: Proceedings of the IEEE/ACM international conference on Automated software engineering. pp. 185–194. ACM (2010)
15. Steel, J., Drogemuller, R., Toth, B.: Model interoperability in building information modelling. Software and Systems Modeling pp. 1–11 (2010)
16. Storey, M.A.D., Fracchia, F.D., Müller, H.A.: Cognitive design elements to support the construction of a mental model during software exploration. Journal of Systems and Software 44(3), 171–185 (1999)
17. Weiser, M.: Program slicing. In: Proceedings of the 5th international conference on Software engineering. pp. 439–449. IEEE Press (1981)