# Kompren: Modeling and Generating Model Slicers

Arnaud Blouin, Benoit Combemale, Benoit Baudry, Olivier Beaudoux

# Kompren: Modeling and Generating Model Slicers*

Arnaud Blouin† Benoît Combemale‡ Benoit Baudry§ Olivier Beaudoux¶

## Abstract

Among model comprehension tools, model slicers are tools that extract a subset of model elements, for a specific purpose. Model slicers provide a mechanism to isolate and focus on parts of the model, thereby improving the overall analysis process. However, existing slicers are dedicated to a specific 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.* modeling for software development or for civil engineering) and for different purposes (*e.g.* monitoring and model comprehension). We detail the semantics of the Kompren language and of the model slicer generator. This provides a set of expected properties about the slices that are extracted by the different forms of the slicer. Then we illustrate these different forms of slicers on case studies from various domains.

*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.

†Arnaud Blouin
INSA Rennes, IRISA/INRIA, Triskell Team, Rennes, France
arnaud.blouin@irisa.fr

‡Benoît Combemale
University of Rennes1, IRISA/INRIA, Triskell Team, Rennes, France
bcombemale@irisa.fr

§Benoit Baudry
INRIA Rennes, IRISA/INRIA, Triskell Team, Rennes, France
bbaudry@inria.fr

¶Olivier Beaudoux
TRAME-ESEO, Angers, France
olivier.beaudoux@eseo.fr

# 1 Introduction

## 1.1 Context

Program slicing [44] is a "*technique for focusing on certain aspects of a program's behavior and removing all other parts of code not concerned with this behavior* [23]". The two major slicing methods are currently static and dynamic slicing. Static slicing is an operation that takes as input slicing criteria, *i.e.* variables and their position in the program to slice. This operation produces as output a slice composed of the statements that *may* have effects on the slicing criteria. The static slicing operation does not execute or interpret the program so that the output slice may not be minimal. For instance, control flows such as `if(foo) then ... else ... endif` are not evaluated to state which conditional branch, *then* or *else*, must be sliced; the whole *if* statement is sliced. Dynamic slicing remedies this drawback by evaluating the programs' statements. The interested reader can refer to [35, 13, 11, 45, 39] for more details on program slicing.

Model slicing is a model comprehension technique inspired by program slicing. The process of model slicing involves extracting a subset of model elements which represent a *model slice*. The model slice may vary depending on the intended purpose. For example, when seeking to understand a large class diagram, it may help to extract the sub-part of the diagram that includes only the dependencies of a particular class. For other comprehension purposes one might want the footprint of model operations [17], or extracting information from several interdependent models [23].

Program slicing transposed to models can be divided into static and dynamic slicing as well. Static model slicing consists in slicing models according to structural criteria. For instance, slicers relying on the MOF (*Meta-Object Facility*[1]) metametamodel will slice the structure of models (classes, properties, *etc.*). Dynamic model slicing considers the behavioral semantics of the input metamodel and requires the execution of the sliced model. For example, slicing an automaton with respect to a specific event as a slicing criterion, consists in extracting a sub-automaton, which reacts to the selected event. In this paper we focus on static slicing of models.

There has been previous work on the definition of model slicers. But all the 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

[1] `http://www.omg.org/mof/`

modeling languages (DSMLs) appear regularly to improve productivity, this becomes an issue: on the one hand, it is not convenient to develop slicers from scratch for each 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 comprehension tools as general purpose languages. Thus it is necessary to develop a generative approach that will automatically build model slicers for new metamodels.

### 1.2 Contributions

In this paper we propose Kompren[2], a DSML to model model slicers for a particular domain (captured in a metamodel). The knowledge gained from practical experience and current model slicers, lead to the design choices of the Kompren language. The primary objective of Kompren is the selection of classes and properties in an input metamodel. Kompren promotes the definition of slicers that slice all necessary elements to make the slice a valid instance of the input metamodel. Kompren also facilitates the relaxation of the conformance required by the input metamodel. Kompren offers a set of language features to generate model slicers that can still be parameterized to process the model slice for a specific purpose. The different characteristics of Kompren tackle two goals for our generative approach: automatically building 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:

1. a language to model model slicers for any metamodel;
2. an illustration of the language expressiveness over three uses cases on model operation analysis, model comprehension, and model monitoring at runtime;
3. a systematic classification of properties one can expect from the model slicers generated by Kompren;
4. an exhaustive classification of the related work on model slicing;
5. a complete set of tools to define and execute model slicers, including editors and a new version of the compiler featuring evaluated performance improvements.

This paper extends our work published at MODELS 2011 [7] with the last three contributions 3, 4, and 5.

### 1.3 Paper Outline

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 describes the Kompren tools provided to users and different benchmarks to analyze the scalability of the implementation. Section 6 demonstrates the expressiveness of Kompren on the three illustrative cases introduced in Section 2. Section 7 discusses the related work on model slicing. Section 8 concludes this work and proposes a research agenda on model slicing.

[2] `http://people.irisa.fr/Arnaud.Blouin/software_kompren.html`

## 2 Heterogeneous Use Cases of Model Slicing

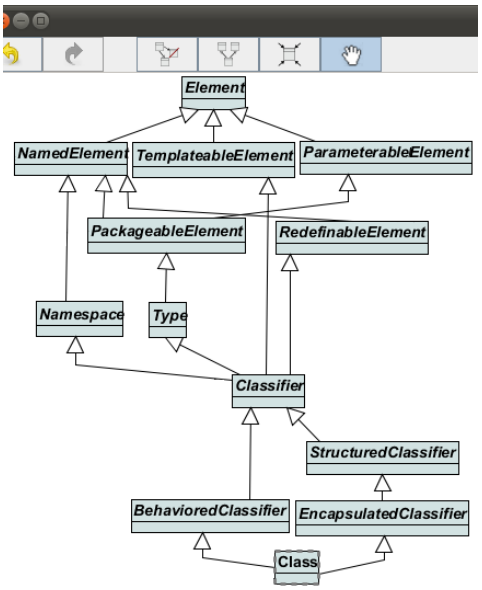
The classical use of model slicing consists in extracting submodels from models by keeping conformance rules. However, as shown in the motivating use cases below model comprehension also requires extracting models which do not satisfy conformance. Still, this extraction can rely on model slicing mechanism.

**Use case 1: Model operation analysis.** Given a model operation on a large metamodel $MM_1$, developers demand 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 the $MM_1$ elements it uses and to get the effective metamodel $MM_2$ [31].

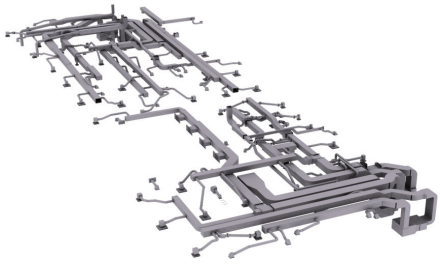
In terms of program slicing this is similar to the technique used in bytecode shrinking. For example, Proguard [3] analyzes Java bytecode to eliminate all classes that are not used.

**Use case 2: Semantic zooming on models.** Several program slicing methods have been used to assist in program comprehension (*e.g.* [6, 29]). Similarly, understanding and manipulating large models require visualization techniques to provide meaningful navigation capabilities [38]. Semantic zooming is a Human-Computer Interaction (HCI) technique that can be applied for this purpose. In contrast to physical zooming that alters the size of objects, semantic zooming changes the type and meaning of information displayed by objects [15]. For instance, as shown in Fig. 1a, semantically zooming on class inheritance extracts superclasses of a given class. We can notice that semantic zooming does not require the output slices to conform to their metamodel; the output slices are not saved as new models, but used by HCI features to perform semantic zooming.
The model slicing applications are not limited to the computer science domain. For example, recent work proposed a model-driven approach in civil engineering for the interoperability and comprehension of building models [36]. This application of MDE is particularly challenging for all MDE tools since the entire model includes more than 5M of model

[3] http://proguard.sourceforge.net/




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



(b) Complex Mechanical Model of a Building, extracted from [36]

Figure 1: Examples of Semantic Zooms

elements. In this context, model slicing is particularly relevant to understand and analyze the model from different perspectives. Stakeholders may need tools to improve their comprehension of the different concerns of the building being designed. In such a context, model slicers can extract information from the whole building model to display different concerns. For example, Fig. 1b shows the mechanical model of a building. Mechanical model stakeholders are eager 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, state-based model stakeholders may want to monitor the current state. Thus dedicated tools need to extract only information relevant to the current state. Such information must be incrementally extracted to improve performance on large models. This use of model slicing is similar to slicing techniques extract the value of variables of a running program for debugging [39].

# 3 Overview

Fig. 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 Fig. 2). A *model slicer model* (MSM) expressed with *Kompren* refers to a set of classes and relations from the *input metamodel* expressed using an object-oriented meta-language (*e.g.,* Ecore, in our case). Instances of the referenced classes and relations will be selected for slicing in the *input model*. Thus MSMM points to Ecore to enable MSM to use Ecore elements from an input metamodel. Because Ecore describes the structure of metamodels, the Kompren model slicers are syntactic. MSMM also points to Kermeta [26], an action language used to specify the behavior of a slicer. *Kompren*'s compiler processes an MSM defined for an *input metamodel* and automatically generates an actual *model slicer function* (MSF).

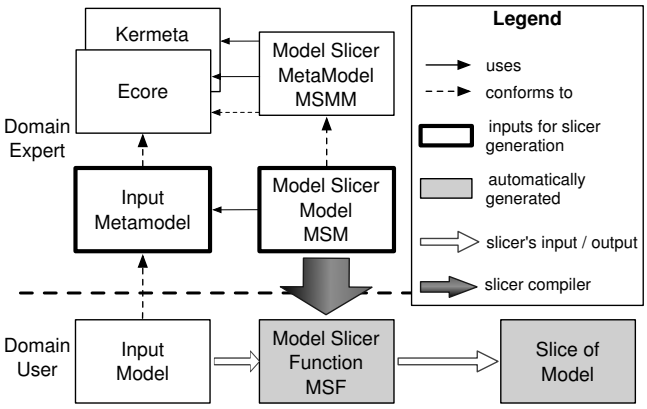



Figure 2: Overview for Modeling Model Slicers with Kompren

In this context, a slicing criterion is a set of model elements that provide the entry point for extracting a model slice. Using Kompren, the MSM specifies the type of the slicing criteria among the classes of the input metamodel. Then, Kompren generates the set of corresponding parameters for the MSF, letting the domain expert specify a slicing criterion to execute the MSF.

This global approach is a two-level generation process: *Kompren*'s compiler generates an MSF, which in turn generates model slices. 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 to model one or several model slicers relevant for this domain. The *domain expert selects the elements in the input metamodel* that will be processed by the model slicer through the MSF.

- **Domain users** create models in the domain. These users, through their modeling activities, can create large instances of the *input metamodel*. At some point these users need to extract slices thanks to the MSF. These *users parameterize the model slicer according to their need and according to the values in the input model*.

# 4 Model-Driven Specification of Slicers

## 4.1 Kompren Features for Model Slicer Generation

A *model slicer* is specified by a *Model Slicer Model* (MSM) and implemented by a *Model Slicing Function* (MSF) generated from the MSM (*cf.* Fig. 3).

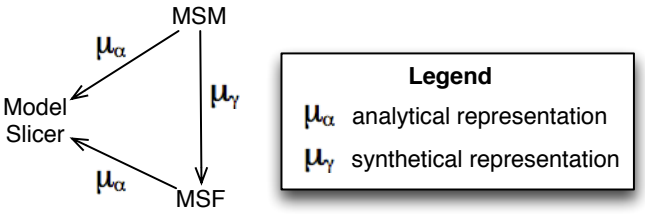



Figure 3: Relationships between Model Slicer, MSM, and MSF according to the analytical and synthetical representations proposed in [27]

An 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, the slicing operation consists in visiting the model for a particular purpose according to the structure of their metamodel and the classes and properties specified in the MSM. Such a processing is performed by a *Model Slicing Function* (MSF) generated from an MSM, and results in a *slice*.

Below, we detail the features offered by Kompren to (i) ease the modeling of MSMs, and to (ii) specify an evaluation mode of an MSF. We also discuss the different properties one can expect on slices generated by the MSF (*cf.* Fig. 4). We use an example to illustrate theses features: the class diagram *input metamodel* (Fig. 5a) and the *input model* shown in Fig. 5b.

Kompren proposes the following constructs to assist the definition of an MSM (left part of Fig. 4):

- **Add a transient opposite property in the input metamodel to ease the slicing.** For example, Fig. 6b is a slice of 5b 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*.

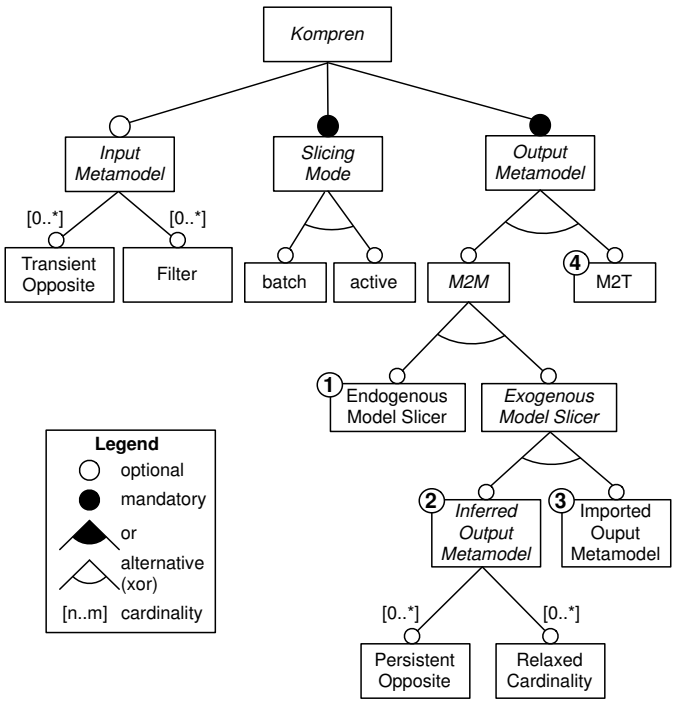



Figure 4: Kompren's Features

- **Add constraints to filter the sliced elements with respect to the *input metamodel*.** For example, Fig. 6c is a slice of 5b composed of the class `A` and of its composite references only. Similarly, Fig. 6d is a slice of 5b that selects `B` and its direct supertypes.

### 4.1.1 Slicing Mode

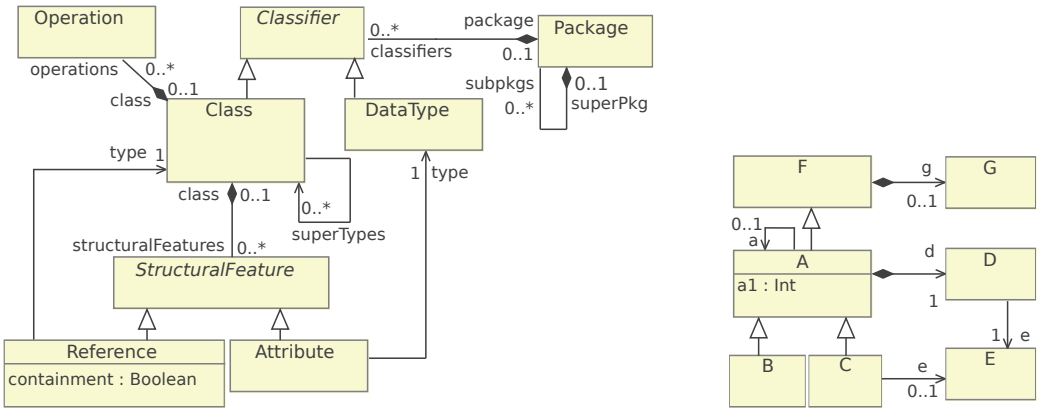
The MSF can be generated from one MSM with one of the following *slicing mode*:

- ***Batch*:** the MSF slices the input model once, when called by the domain user (*cf.* use cases 1 and 2);
- ***Active*:** the MSF automatically updates the slice each time the input model changes (*cf.* use case 3).

### 4.1.2 Slicing Output Formats

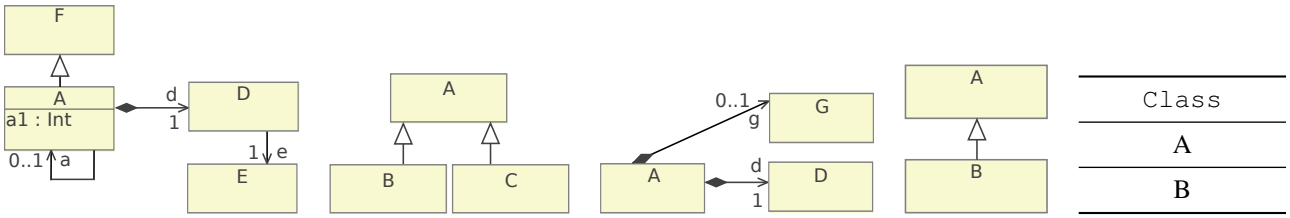
According to the specified MSF and the selected options, the model slicers will generate model slices, which have different properties. In the following we list the properties one can expect from a model slice (right part of Fig. 4, feature model below '*Output metamodel*').

First, the MSF can provide a resulting slice as a new model satisfying all the structural constraints imposed by the input metamodel. In such a case the slice is a valid instance of the input metamodel, and we call the corresponding model slicer an *endogenous model slicer* . For example, Fig. 6a is a slice of Fig. 5b that includes only *A* and *F*, as well as the mandatory classes *D* and *E* to satisfy the conformance with the *input metamodel*.



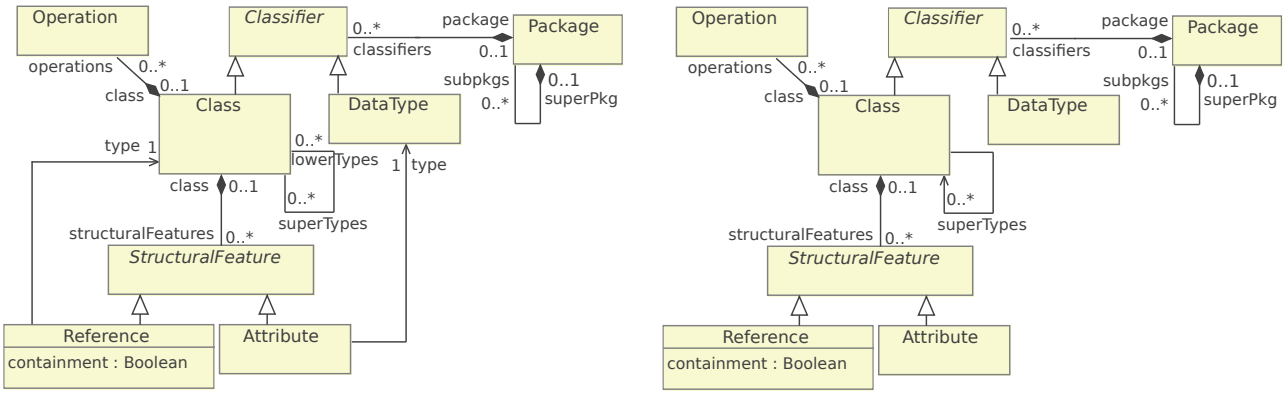

(a) Input Class Metamodel (b) Input Class Model

Figure 5: Class Model Example




(a) Strict Mode (b) Opposite (c) Constraint (d) Radius (e) Database Slice

Figure 6: Slices of the Class Model given in Fig. 5b




(a) Metamodel 5a augmented with the opposite `lowerTypes` (b) Metamodel 5a Constrained by removing the relations 1..1 `type`

Figure 7: Inferred Output Metamodels

It is also possible to use Kompren to generate model slicers that relax the conformity constraint on slices in exchange of additional features for model slicer modeling.

Model slicers providing a slice conforming to a different metamodel than the input metamodel are called *exogenous model slicer*. We distinguish two categories of *exogenous model slicers*:

- Exogenous model slicers that generate a slice conforming to a metamodel inferred by analyzing the MSM and adapting the *input metamodel* (*cf.* ② in Fig. 4).
  - **Add a persistent opposite property in the output metamodel.** For example, Fig. 6b could be saved with the opposites of the `superTypes` property used to ease the slicing (*e.g.* to facilitate its navigation according to this property in a later processing). In such a case, the *input metamodel* must be enriched with the opposite properties to save the resulting slice (*cf.* Fig. 7a).
  - **Add constraints to filter the sliced elements removing elements required to conforms to the *input metamodel*.** For example, Fig. 7b shows a constrained version of the metamodel of Fig. 5a where the two relations *type* (cardinality $[1]$) were removed.
- Exogenous model slicers that generate slice conforming to a metamodel explicitly imported by the domain expert in its MSM definition (*cf.* ③ in Fig. 4). For example, Fig. 6e is an RDBMS[4]-based slice extracted from Fig. 5b by selecting the class *B* and its direct classes (*A*).

In addition to *Model to Model* (M2M) model slicers, Kompren also supports the generation of MSF that produce texual slices. We call these slicers *Model to Text* (M2T) slicers. These can be used to print information about the sliced model elements or to notify external tools about the slice.

All the previous expected features were considered in the design of the Kompren language. We present respectively in the remainder of this section the abstract syntax, concrete syntax, and semantics embedded into the compiler of the Kompren language.

## 4.2 Kompren Abstract Syntax

The metamodel shown in Fig. 8 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 are 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[5] (*uriMetamodel*). Optional *SlicedElements* (*i.e. isOption* is true) are options of the generated MSF. This lets the domain user choose whether an element must be considered during the slicing.

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 (*Object Constraint Language*[6]) as follows:

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

Similarly, a *SlicedProperty* refers to a property (*EStructuralFeature*) in the *input metamodel* (*domain*). 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 type of the *src* and *tgt* iterators is respectively the source and the target class of the property:

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

In addition, a sliced property may specify an *OppositeCreation* to define that an opposite must be created on the targeted *domain*. The name of this new opposite is given by the attribute *name*. By default, such opposites are used as helpers for exploring the input metamodel and are not serialized in the output slice. But developers may want to serialize some of the opposites; in such a case, the attribute *transient* of a given *OppositeCreation* must be set to *false* and an output metamodel, augmented with the selected opposites, will be inferred.

We assume in this paper an *input metamodel* defined with an existing object-oriented metamodeling language. In our experiments we use the Ecore metamodeling language provided by the *Eclipse Modeling Framework*[7] whose required elements are imported in the package *ecore*. In Ecore, a class and a property are identified by respectively the classes *EClass* and *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 [26] whose required elements are imported in the package *kermeta*. Another action language could be easily considered in Kompren.

By default, the model slicer can be exogenous or endogenous depending on the behavior defined by the developers. The expressions *onStart* and *onEnd* are used to add a par-

[4] *Relational DataBase Management System*

[5] *Uniform Resource Identifier*

[6] `http://www.omg.org/spec/OCL/`

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

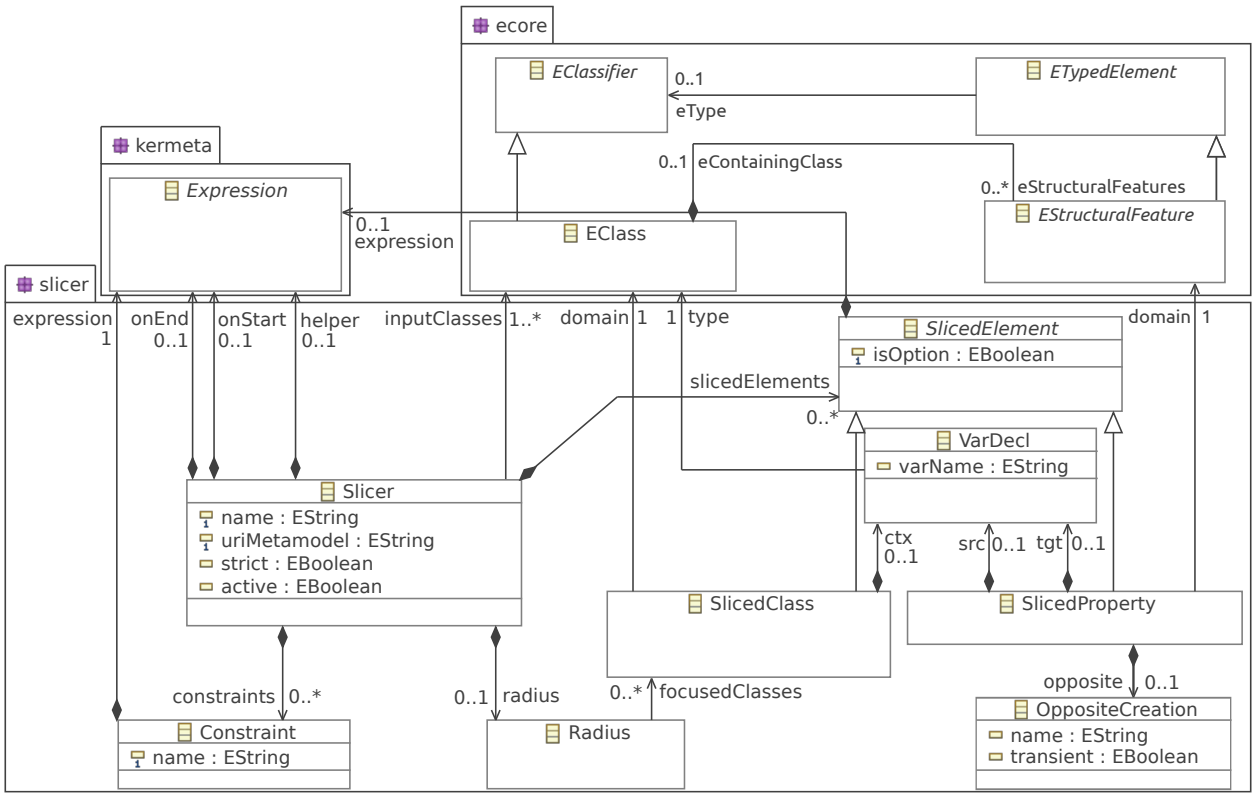



Figure 8: Model Slicer Metamodel

ticular behavior in the MSF. These expressions are respectively applied before and after the visit of the *input model* (*e.g.* to import the required output metamodel and save the resulting slice respectively). Expressions defined to bring executability to slicers may require classes provided by third party libraries, attributes, or operations needed to the slicing process. Thus the domain expert can specify a *helper* that will contain this information. The attribute *strict* (in *Slicer*) defines whether the slicer must be endogenous (*cf.* Section 4.1.2). By setting the attribute *strict* to true, the model slicer bypasses the filters expressed into the MSM to ensure slices conform to the *input metamodel*.

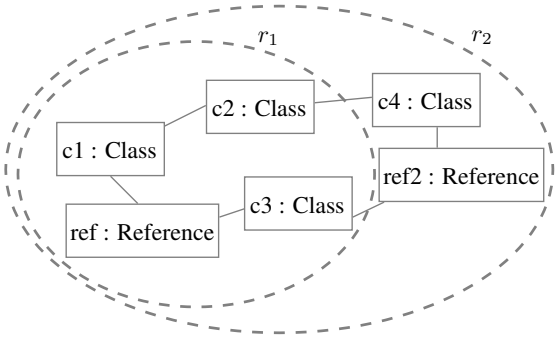



Figure 9: Example of the Radius Process

The *radius* and the *constraints* can be used to filter the sliced elements in the *input model*. The *radius* sets in the MSM the *focusedClasses* for which the MSF should be limited to a selection within a given radius. Starting at 0, a value is incremented on each visited class instance focused by the radius. The slicing process stops when no elements can be sliced anymore or when this value is greater than the radius given as parameter. Fig. 9 shows an example of the radius process where the focused class is *Class* and the slicing criterion $c_1$. Each dashed ellipse shows the sliced instances for a specific radius value: $c_1$, $c_2$, $c_3$, and $ref$ are sliced when the radius equals 1; $ref$, $ref_2$, and $c_1$ to $c_4$ are sliced when the radius equals 2. The radius is defined by the domain expert for a sliced class, and its value must be specified by the domain user as a parameter of the MSF. The focused classes must be included in the sliced classes that can be formalized using OCL as follows:

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

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 the slicing criteria that the MSF will take as input to start the slicing.

Finally, the attribute *active* permits to specify whether the MSF must be executed as a batch or as an active process. By default, the generated MSF is a batch process executed a single time on the input model. By setting the attribute

*active* to true the generated MSF is executed a first time, and then observes modifications applied on the input model 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 non-strict MSM *ClassModelSlicer* (*cf.* line 1), for the metamodel in Fig. 5a (*cf.* line 2). The classes of the instances used to launch the *Model Slicing Function* (MSF) are declared on line 3.

Thereafter, line 6 specifies a sliced class while lines 7 to 10 specify the sliced properties. An expression defined for the sliced class `Class` is described on line 6 where *cl* refers to the context of the sliced class. An optional property is illustrated on line 7 with the keyword `option`. An opposite to a property is defined with the keyword `opposite` as shown on line 8 where *lowerTypes* is the name of the opposite.

Line 4 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 on line 5. Lines 11 to 13 illustrate the definition of the preprocessing, the postprocessing, and the helper of the slicer.

```
1  slicer active ClassModelSlicer {
2   domain: "platform:/resource/classModel.ecore"
3   input: Class
4   radius: Class
5   constraint: Reference.containment
6   slicedClass: Class cl[[ stdio.writeln(cl.name) ]]
7   slicedProperty: Class.superTypes option
8   slicedProperty: Class.superTypes opposite(lowerTypes)
9   slicedProperty: Class.structuralFeatures
10  slicedProperty: Reference.type
11  onStart [[ stdio.writeln("Starting slicing") ]]
12  onEnd   [[ stdio.writeln("Ending slicing") ]]
13  helper  [[ /* Definition of the helper */ ]]
14 }
```

### 4.4 Semantics

As defined in Fig. 2, model slicer models (MSM) are compiled into model slicer functions (MSF). This compilation produces Kermeta programs composed of three parts. The first part augments the input metamodel with required information. This information is the opposites added to the input metamodel and methods required by the visitor 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 performs a static analysis of the MSM to infer whether a new metamodel is required as output metamodel (usually to relax some cardinalities).

Finally, the compiler generates the MSF. The preprocessing (*onStart*) and the postprocessing (*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 (the slicing criteria); *radius:Integer* specifies the slicing radius; *composition:Boolean* is a constraint that restricts the slicing of references to composite references.

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

Once generated, the MSF 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 (*i.e.* the slicing criteria) given as parameters. Each of these instances is visited. Visiting an instance or a property consists in executing the associated behavior, *i.e.* the corresponding Kermeta expression defined by the domain expert. 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. In case of a strict slicer, the MSF also slices the required model elements to conform to the input metamodel, and automatically add all the sliced instances into a new model.

Because Kermeta does not support observability of Ecore models, *active* slicers are based on the *ActiveKermeta* toolkit [5]. *ActiveKermeta* adds observability to Ecore models through the operations *c.added{e | ...}*, *c.removed{e | ...}*, and *c.updated{p, e | ...}*. These operations register a function respectively invoked when the element *e* is added to collection *c*, removed from collection *c*, or when the element *p* is replaced by the element *e* into *c*.

## 5 Tooling

### 5.1 Domain Expert

We provide domain experts with a comprehensive set of prototyping tools to develop model slicers[8]. These tools are composed of two editors and of a compiler running on the top of Eclipse[9] and Scala[10].

#### 5.1.1 The Kompren Editors

Two editors are provided for editing MSMs. The first one is a customized Ecore reflexive editor that edits MSMs through a tree-based presentation. The second editor is a textual editor based on our proposed concrete textual syntax (see Fig. 10 for a screenshot of this editor). This editor provides auto-completion, serializes as Ecore models, and compiles MSMs in MSFs.

[8] http://people.irisa.fr/Arnaud.Blouin/software_kompren.html
[9] http://www.eclipse.org/
[10] http://www.scala-lang.org/

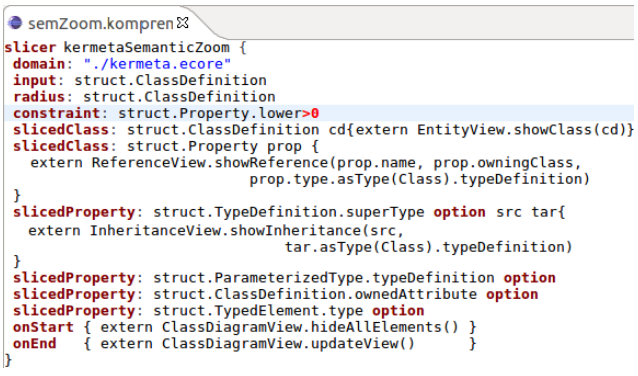



Figure 10: Kompren Editor in the Eclipse Environment

#### 5.1.2 The Kompren Compiler

The MSM to MSF compiler has been developed using Kermeta 2 running on the top of Scala (Kermeta 2 programs are compiled into Scala, then into Java byte-code). We performed benchmarks on several use cases we developed to study the compilation time. These uses cases have different levels of complexity as described in Table 1. This table describes the characteristics of the use cases: $\mathcal{S}_c$ and $\mathcal{S}_p$ respectively define the number of classes and properties of the input metamodel; $\mathcal{S}_{sc}$ and $\mathcal{S}_{sp}$ respectively define the number of the sliced classes and properties; $\mathcal{P}_k$ enumerates the Kompren's features used; $\mathcal{T}_c$ gives the average compilation time (in second) for 100 executions of the use cases. The experiments described in this section have been performed on Linux using a laptop with a Core2Duo at 3.06GHz and 4Gb of RAM, Scala 2.9.0, and Java 7.

| Use cases | $\mathcal{S}_c$ | $\mathcal{S}_p$ | $\mathcal{S}_{sc}$ | $\mathcal{S}_{sp}$ | $\mathcal{P}_k$ | $\mathcal{T}_c$ |
|---|---|---|---|---|---|---|
| Super-classes | 1 | 1 | 1 | 1 | not strict | 0.189 |
| State-machine | 8 | 7 | 7 | 3 | strict | 0.209 |
| Semantic Zooming Ecore | 20 | 51 | 9 | 10 | not strict, opposite, radius | 0.371 |
| Kermeta Operation Analysis | 73 | 95 | 1 | 29 | not strict | 0.503 |
| UML Class Diagram Extraction | 246 | 769 | 16 | 21 | strict | 2.026 |

Table 1: Several MSM Compilation Benchmarks

For the simplest use case $\mathcal{T}_c$ equals 0.189s. For the most complex one $\mathcal{T}_c$ equals 2.026s. Because finding the mandatory elements to slice requires extra computations, the *strict* property complexifies the MSFs generation. Large input metamodels increase the computations as well.

### 5.2 Domain User

Once defined by domain experts, MSF can be used by domain users. As the compiler, MSFs are Kermeta 2 programs running on the top of Scala. MSFs take as inputs the slicing criteria, the options, the constraints, and the radius value as defined in the MSMs. MSFs produce as output either a sliced model when the model slicer is strict, nor results as defined by domain experts.

MSFs can be integrated into applications. For instance, our tool that visualizes metamodels uses an MSF using the Kermeta metamodel as input metamodel.

| $\mathcal{S}_m$ | $\mathcal{T}_e$ |
|---|---|
| 100 | 0.023 |
| $10^3$ | 0.031 |
| $10^4$ | 0.141 |
| $10^5$ | 0.425 |
| $10^6$ | 9775.51 |

Table 2: Several Execution Benchmarks of the State Machine Slicer

In our current prototype, the slicing process is based on the visitor design pattern using a deep-first search algorithm. Following Table 1, Table 2 gives the slicing execution time $\mathcal{T}_e$ (in second) for the state-machine slicing example presented in Table 1. The benchmarks have been performed using input models of different sizes: $\mathcal{S}_m$ refers to the number of elements that the models contain. The goal of this experiment is to demonstrate that Kompren can slice large models in a reasonable time. For the sizes from 100 to $10^5$, 100 models (more precisely 100 connected state-machines) were randomly generated. $\mathcal{T}_e$ is the average execution time for slicing these models. Because of technical limits, only a single model was randomly generated for the size $10^6$. In this case, $\mathcal{T}_e$ is the average execution time for slicing this model 100 times. The goal of this strict slicer is to slice the whole input model using as input the initial state. For the smallest size, 100, $\mathcal{T}_e$ is 0.023. For the larger size, $10^6$, $\mathcal{T}_e$ is 9775.51. Because of the model loading and saving operations, the progression of the execution time over the different sizes is not linear. The materials used for these benchmarks are available on the Kompren website[11].

### 5.3 Threats to Validity

The benchmarks performed on non-strict Kompren model slicers strongly depend on the Kermeta expressions, defined by the developers, used in the slicer. Optimizations may be applied on our current prototyping implementation to improve these benchmarks.

[11] http://people.irisa.fr/Arnaud.Blouin/software_kompren.html

The slicing execution time $\mathcal{T}_e$ strongly depends on the parameters of the model slicers. For instance, the definition of an opposite in a model slicer will increase the execution time by 2 at least: before the slicing operation, the opposite must be integrated into the input model; this operation requires the whole exploration of the input model.

# 6 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.

## 6.1 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 operation [17]. In this section, we use Kompren to model the footprint generator proposed by Jeanneret *et al.* [17] and the metamodel pruner proposed by Sen *et al.* [31]. This use case illustrates the ability of Kompren to: ease the slicer definition process; combine several Kompren model slicers to perform a task not related to model slicing.

This model operation analysis 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 first slicer extracts the list of $MM_1$ elements used by the operation. Because such a slice does not conform to $MM_1$, the slicer is not strict. 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 6 to 16). The result of the slicing function will be the list of classes used in the operation (line 5). This list is defined in the helper (line 18). By default all the classes that can come from either $MM_1$ or $MM_{op}$ are explored. Because only the classes from $MM_1$ must be stored, a helper is defined to select them (lines 19 to 25).

```
1  slicer OperationStaticAnalysis {
2   domain: "./kermeta.ecore"
3   // The model operation to analyse.
4   input : struct.ModelingUnit
5   slicedClass:struct.ClassDefinition cd[[addClassDef(cd)]]
6   slicedProperty: struct.ModelingUnit.packages
7   slicedProperty: struct.Package.ownedTypeDefinition
8   slicedProperty: struct.ClassDefinition.ownedOperation
9   slicedProperty: struct.ClassDefinition.ownedAttribute
10  slicedProperty: struct.Operation.ownedParameter
11  slicedProperty: struct.TypedElement.type
12  slicedProperty: struct.ParameterizedType.typeDefinition
13  slicedProperty: struct.Operation.body
14  slicedProperty: behavior.VariableDecl.type
15  slicedProperty: behavior.Block.statement
16  //... 29 properties of MM_op are sliced.
17  helper [[
18   reference metamodelClassesUsed : ClassDefinition[0..*]
19   reference inputMetamodel : ModelingUnit
20   //... Load of the input metamodel.
21   operation addClassDef(cd: ClassDefinition): Void is do
22    if(inputMetamodel.contains(cd)) then
23     metamodelClassesUsed.add(cd)
24    end
25   end]]
26  }
```

The second slicer, modeled as follows, uses the footprint computed by the first one. This slicer is defined as *strict* (line 1) to create an output model that is an endogenous 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*.

```
1  slicer strict MetamodelFootprintExtraction {
2   domain: "./kermeta.ecore"
3   input : struct.ClassDefinition
4   slicedClass: struct.ClassDefinition
5   slicedClass: struct.Property
6   slicedClass: struct.Operation
7   slicedProperty: struct.ClassDefinition.ownedAttribute
8   slicedProperty: struct.ClassDefinition.ownedOperation
9   slicedProperty: struct.Operation.ownedParameter
10  slicedProperty: struct.TypedElement.type
11  slicedProperty: struct.TypedDefinition.superType
12  slicedProperty: struct.ParameterizedType.typeDefinition
13 }
```

These Kompren model slicers are smaller than the Jeanneret's and Sen's model slicers: around 70 Kompren LoC compared to 1200 Kermeta LoC for both the static metamodel footprinting and the metamodel pruner. The number of generated Scala LoC when compiling a Kompren model into an executable slicer (a Scala program) is another relevant metric. This use case generated around 6700 of Scala LoC [12]. Because Kompren is a DSL dedicated to the definition of model slicers, Kompren hides some technical details such as the *Visitor* pattern. By opposition, Kermeta and Scala are GPLs (*General Purpose Language*) that require the explicit definition of such technical details. This difference can explain the gap in terms of LoC differences between the slicers.

## 6.2 Bringing Semantic Zooming 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 not defined as *strict*. It takes as input instances of *ClassDefinition* (line 3) selected by users using the UI. As shown in Fig. 11, the UI displays classes, inheritances, and properties. At the beginning of the slicing all these model elements are hidden (line 23). Then when model elements are sliced, the UI is notified that these elements must be shown

[12] This number can certainly be reduced by optimizing the compiler.

(lines 7, 9, and 17). At the end of the slicing, the UI is updated to perform the graphical changes (line 24). Some properties must be explored to access the instances to slice (lines 14 to 22). 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 kermetaSemanticZoom {
2   domain: "./kermeta.ecore"
3   input: struct.ClassDefinition
4   radius: struct.ClassDefinition
5   constraint: struct.Property.lower>0
6   slicedClass: struct.ClassDefinition cd[[
7    extern EntityView.showClass(cd) ]]
8   slicedClass: struct.Property prop {
9     extern ReferenceView.showReference(
10      prop.name, prop.owningClass,
11      prop.type.asType(Class).typeDefinition)
12  }
13  slicedProperty: struct.TypeDefinition.superType option
14   src tar[[
15     extern InheritanceView.showInheritance(
16       src, tar.asType(Class).typeDefinition)
17  ]]
18  slicedProperty: struct.ParameterizedType.typeDefinition
19    option
20  slicedProperty: struct.ClassDefinition.ownedAttribute
21    option
22  slicedProperty: struct.TypedElement.type option
23  onStart [[ extern ClassDiagramView.hideAllElements() ]]
24  onEnd   [[ extern ClassDiagramView.updateView()      ]]
25 }
```

The UI shown in Fig. 11 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 whether 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.

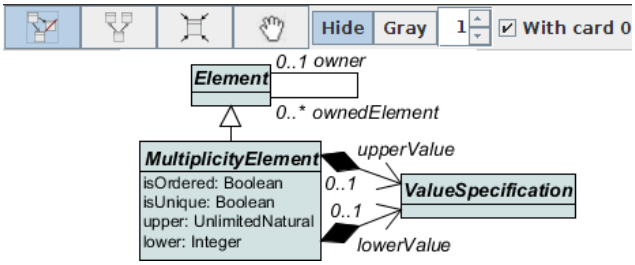
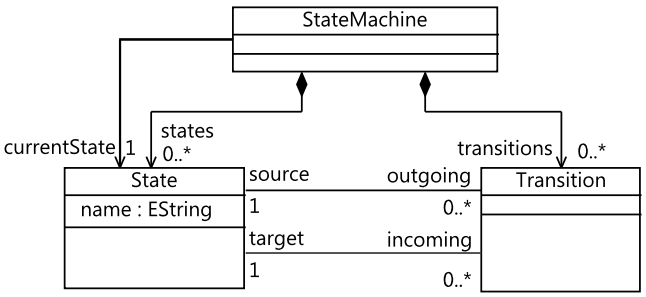



Figure 11: Class Diagram Visualizer Providing Semantic Zooming Features

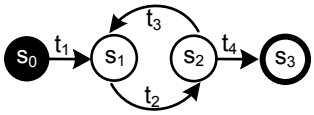
## 6.3 Monitoring State-machines at Runtime

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. In such a context, a slice can be used to observe how a specific sliced part of a larger model evolves.

Fig. 12a describes a basic state-machine metamodel. The root class *StateMachine* specifies all the *states* and *transitions* defined within the state-machine. A *Transition* links a *source* state and a *target* state; conversely, a *State* is linked to




(a) A State-Machine Metamodel




(b) A State-Machine Model

Figure 12: A State-Machine Example

other states throughout *incoming* and *outgoing* transitions. The relation *currentState* defines the current state during the execution of the state-machine.

Fig. 12b shows a state-machine composed of four states $s_0$ to $s_3$ and four transitions $t_1$ to $t_4$. Slicing such a model at runtime consists in capturing the evolution of a model slice while the state-machine is running, *i.e.* slicing changes of the current state.

The following Kompren code slices the current state of a state-machine and displays its name on changes. The slicer first displays the name of the initial state. Whenever the *currentState* relation *cs* is updated the anonymous function given by the *updated* method is invoked. In this method *prev* and *next* are the previous and the new states contained in *cs* respectively. Running the *CurrentStateSlicer* with the sequence of transitions $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_2 \rightarrow t_4$ results in displaying $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3$.

```
1  slicer active CurrentStateSlicer {
2    domain: "platform:/resource/statemachine.ecore"
3    input: StateMachine
4    slicedClass: PseudoState
5    slicedClass: State
6    slicedProperty: StateMachine.currentState cs [[
7     stdio.write(cs.name) // Printing the initial state
8     // Printing the state's name on update
9     cs.updated{prev, next | stdio.write("->" + next.name)}
10   ]]
11 }
```

As this example illustrates, Kompren active slicers are based on Active Kermeta that makes the four collections provided by Kermeta (*set*, *oset*, *seq*, and *bag*) observable [5]. Relation *currentState* is implemented using an Active Kermeta *set* having its cardinality restricted to [1..1]. Using collections for relations with cardinality [0..1] or [1..1] is mandatory for observing their content. Two others methods, *added* and *removed*, are provided by Active Kermeta collections to respectively observe additions and removals into collections.

| Approaches \ Properties | *Slicing Mode* | | *Slicing Process* | | *Slicer* | | | *Slice* | KOMPREN*able* | *Metamodel* | *Usage* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Batch** | **Active** | **Syntactic** | **Semantic** | **M2T** | **Endogenous** | **Exogenous** | **Struct. Mod.** | | | |
| Feature Model Slicing [1, 2] | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | Feature Model | Separation of Concerns |
| Model Projection [3] | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | State-Based Model | correctness/efficiency |
| UMLSlicer [4] | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | UML Metamodel | Modularization |
| UML Statechart [8] | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | UML Statechart | Reactive Systems |
| SafeSlicer [10] | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | SysML | Safety |
| Feature Model Slicing [16] | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✓ | Feature Model | Visualization |
| Metamodel Footprint [17] | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✓✗ | Ecore-based | NA/A (generic) |
| Context-Free UML Slicing [18] | N/A | N/A | ✓ | ✗ | ✗ | N/A | ✗ | ✗ | N/A | UML Class Diagram | Sub-Models Extraction |
| Sub-Model Lattice [19] | ✓ | ✗ | ✓ | N/A | ✗ | ✓ | ✗ | ✗ | N/A | Ecore-based | Comprehension |
| Software Architecture [20] | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | N/A | Software Architecture |
| EFSM Slicing [21] | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✓ | State-Based Model | Size Reduction |
| DSUAM [23] | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | UML | Separation of Concerns |
| UML Slicing [24] | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | UML | Comprehension |
| UML Activity Diagram [30] | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | UML Activity Diagram | Test Case Generation |
| Metamodel Pruning [31] | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | Ecore-based | Static Analysis |
| UML/OCL Slicing [32, 33, 34] | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | UML/OCL | Verification |
| Model Transformation [40] | ✓ | ✗ | ✓ | ✓ | ✗ | N/A | ✗ | ✗ | ✗ | Viatra2 Transformation | Program Slicing |
| UML Statechart [43] | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | N/A | ✗ | UML Statechart | Model Checking |
| Behavior Tree Slicing [46] | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | N/A | ✗ | Behavior Tree | Model Checking |
| Software Architecture [47] | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | N/A | Software Architecture |

Table 3: Model Slicing Related Work Classification

# 7 Related Work

We classified in Table 3 the main related work using the slicer's properties proposed in Fig. 4 (*slicing mode*, *slicing process*, and *slicer*) supplemented with four others properties: *Slice*. States whether the output slices can be structurally modified compared with the input models; KOMPREN*able*. Defines whether the slicing method can be done using Kompren; *Metamodel*. The supported input metamodel; *Usage*. The context of use of the slicing method.

To our best knowledge, Zhao [47] was the first to use program slicing concepts at a higher level than code. He uses a syntactic, batch, and endogenous slicing algorithm to slice software architectures. Following this work, Kim *et al.* [20] bring semantic slicing to software architecture.

Slicing state-based models has been widely tackled in the literature [14, 21, 41, 22, 3, 43, 8, 25]. Koren *et al.* [21] introduce a batch and endogenous slicing method that uses dependency graphs (data and control dependencies) derived from the state-based models to slice. This method does not evaluate transitions' condition and is thus syntactic. This method provides a post-process step that merges states to reduce the size of the slices.

Androutsopoulos *et al.* [3] propose different finite state-based model slicing algorithms. Their basic slicer removes a set of transitions to ignore and useless states. 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.

Acher *et al.* [1, 2] propose a batch feature model slicer. The slicing process is both semantic and syntactic: the cross-cutting constraints are statically analyzed to define features that must be or cannot be sliced. The output slices still conform to their metamodel but may structurally differ; the feature model and its cross-cutting constraints are first transformed into predicates for analysis. These predicates are then transformed in a sliced feature model.

Hubaux *et al.* [16] slice feature diagrams to design three different views of an input diagram. The sliced diagrams do not keep the same structure as the input diagram. This approach does not consider cross-cutting constraints and is thus syntactic.

Kelsen *et al.* [19] propose an approach for decomposing models into sub-models to tame the complexity of large models. This approach shares similarities with ours since they are both not dedicated to a unique DSML and they can extract sub-models of interest that still conform 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.

Jeanneret *et al.* [17] introduce a method to statically or dynamically extract model footprints. As discussed in Section 6.1 the static footprinting can be done using Kompren. But the dynamic footprinting refers to dynamic slicing concepts that Kompren does not support yet.

Similarly, Ujhelyi *et al.* [40] develop a dynamic slicer for model transformations. The goal of this slicer is two-fold: providing transformation slices depending on the slicing criterion; providing model slices (model footprints) composed the model elements used by the transformation. Backtraces of the execution of the transformation are used to slice the statements used during the execution. Thus this approach is both syntactic and semantic. No information is provided regarding the conformance of the output slices toward their metamodel.

Sen *et al.* [31] present an approach for pruning metamodels. The proposed pruner takes as input slicing criteria, *i.e.* classes, operations, *etc.* of the metamodel to slice. The pruner produces as output slices that satisfy all the structural constraints imposed by the input metamodel. Such a pruner is thus a slicing operation strictly endogenous, syntactically based and is a batch process.

As state-based models, UML is widely tackled in the literature [43, 18, 4, 32, 33, 34, 24, 23, 8]. Shaikh *et al.* [32, 34, 33] use model slicing for verification purpose. The goal of this approach is to check whether an input UML model supplemented with OCL constraints has legal instances. OCL constraints are thus analyzed and interpreted to identify which model elements are constrained.

Closely to Sen's work, Bae *et al.* [4] develop a tool, UML-Slicer, to slice to UML metamodel. As Kompren this tool slices using the structure of the model. But because UML-Slicer does not provide radius and constraint features, it is less expressive than Kompren.

Wang *et al.* [43] introduce a method to reduce the state space during the model checking of UML statecharts. The proposed slicing process is semantically based and exogenous: the output slices are not UML statechart models but extended hierarchical automata (EHA) used by the model checker.

Lano *et al.* [24] present slicing techniques dedicated to UML models. Using these techniques the output slices may structurally differ from the input model without modifying their semantics.

Lallchandani *et al.* [23] 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.

Samuel *et al.* [30] describe a UML activity diagram slicing technique dedicated to the generation of test cases. The input diagram is first converted into a flow dependency graph to be then sliced and be used for the test case generation process.

Yatapanage *et al.* introduce a slicing technique to reduce behavior tree models prior to verification. This batch process uses the semantics of the input model to perform the slicing and produces endogenous output slices.

Fal *et al.* [10] propose a batch model slicer dedicated to the slicing of SysML (*Systems Modeling Language*[13]) models related to safety requirements. The most important step in this approach is the pre-process step that maps a requirements model to a SysML model. Once the mapping established the slicing process operates on the structure of the joined models. Requirements are used as input of the slicer and their related SysML elements that conform to the SysML metamodel.

Clark [9] introduces a model slicing theory that can be used to implement the slicing function as model transformations. This theory is based both on the syntactic and the semantics of the targeted language.

Obeo Designer[14] offers the possibility to easily design 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.

# 8 Conclusion and Future Work

## 8.1 Contributions

Many recent work inspired by program slicing [44] have proposed operations that extract sub parts of models for different purposes [32, 24, 19, 17]. 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 model slicers for domain-specific metamodels. We develop a two-level generative approach on the basis of Kompren: Kompren's compiler processes Kompren models to automatically generate a model slicer function; this function 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 tools. We propose an exhaustive state-of-art on model slicing. 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.* [17] and by Sen *et al.* [31] 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.

[13] `http://www.sysml.org/`

[14] `http://obeo.fr/pages/obeo-designer`

## 8.2 Research Agenda

As an immediate future work, we expect to provide empirical evidence of the Kompren usability elaborating a qualitative evaluation. This will require a study with users to evaluate the qualitative benefits of Kompren w.r.t the construction of slicers with general purpose languages.

Then our perspectives are twofold. First, we aim at supporting the definition of *generic model slicer models*. Currently, the definition of MSMs relies on a specific input metamodel. For instance, one can define a MSM based on the Ecore metamodel to slice class models. However, slicing the class model in a UML class diagram, requires defining a new MSM. But Ecore and UML Class diagram share the concept of class model and relations. MSM definitions could thus be more generic by taking as input not a metamodel (Ecore, UML) but a concept (class model), *i.e.*, a model type [37].

Second, Kompren slicers are currently static since they are based on the structure defined by the input metamodel. A next step of our work will go toward dynamic model slicing. "*While static slicing computes slices with respect to any execution, dynamic slicing computes slices with respect to a particular execution*" [35]. But while all programs are executable, all models are not. Thus, it must be identified: the different kinds of models that can be dynamically sliced; the additional information to use for specifying dynamic model slicers (action languages, *etc.*); the different applications of dynamic model slicing (executable model debugging, *etc.*). Then, other forms of dynamic slicing could be studied, such as conditioned [28], quasi-static [42], and simultaneous slicing [12], that compute slices with respect to a particular set of executions.

# References


[1] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. Slicing feature models. In *26th IEEE/ACM International Conference On Automated Software Engineering (ASE'11)*. IEEE/ACM, 2011.

[2] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. Separation of Concerns in Feature Modeling: Support and Applications. In *Aspect-Oriented Software Development(AOSD'12)*. ACM Press, 2012.

[3] Kelly Androutsopoulos, David Binkley, David Clark, Nicolas Gold, Mark Harman, Kevin Lano, and Zheng Li. Model projection: Simplifying models in response to restricting the environment. In *International Conference on Software Engineering (ICSE'11)*, 2011.

[4] Jung Ho Bae and Heung Seok Chae. UMLSlicer: A tool for modularizing the UML metamodel using slicing. In *8th IEEE International Conference on Computer and Information Technology (CIT)*, pages 772 – 777, 2008.

[5] O. Beaudoux, A. Blouin, O. Barais, and J. M. Jézéquel. Active operations on collections. In *MoDELS'10: Proc. of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 91–105, 2010.

[6] David Binkley, L. Ross Raszewski, Christopher Smith, and Mark Harman. An empirical study of amorphous slicing as a program comprehension support tool. In *Proceedings of the 8th International Workshop on Program Comprehension*, pages 161–170. IEEE Computer Society, 2000.

[7] Arnaud Blouin, Benoît Combemale, Benoit Baudry, and Olivier Beaudoux. Modeling model slicers. In *ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, volume 6981, pages 62–76. Springer Berlin / Heidelberg, 2011.

[8] Miao Chunyu and Zhao Jianmin. Dynamic slicing of statechart specifications for reactive systems. In *Intelligent Computation Technology and Automation (ICICTA)*, volume 1, pages 110–116, 2011.

[9] Tony Clark. A general model-based slicing framework. In *Proc. of the Workshop on Composition and Evolution of Model Transformations*, 2011.

[10] Davide Falessi, Shiva Nejati, Mehrdad Sabetzadeh, Lionel Briand, and Antonio Messina. SafeSlice: a model slicing and design safety inspection tool for SysML. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13)*. ACM, 2011.

[11] Keith Gallagher and David Binkley. Program slicing. In *In Proceedings of Frontiers of Software Maintenance*, 2008.

[12] Robert Hall. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Automated Software Engineering*, 2:33–53, 1995.

[13] Mark Harman and Robert Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[14] M.P.E. Heimdahl, J.M. Thompson, and M.W. Whalen. On the effectiveness of slicing hierarchical state machines: a case study. In *Proceedings of the 24th Euromicro Conference*, volume 1, pages 435–444, 1998.

[15] Ivan Herman, Guy Melançon, and M. Scott Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Trans on Visual Comput Graph*, 6:24–43, 2000.

[16] Arnaud Hubaux, Patrick Heymans, Pierre-Yves Schobbens, Ebrahim Khalil Abbasi, and Dirk Deridder. Supporting multiple perspectives in feature-based configuration. *Software and Systems Modeling*, 2012.

[17] Cédric Jeanneret, Martin Glinz, and Benoit Baudry. Estimating footprints of model operations. In *International Conference on Software Engineering (ICSE'11)*, 2011.

[18] Huzefa Kagdi, Jonathan I. Maletic, and Andrew Sutton. Context-free slicing of UML class models. In *Proc. of the IEEE International Conference on Software Maintenance*, pages 635–638, 2005.

[19] Pierre Kelsen, Qin Ma, and Christian Glodt. 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*, pages 171–185, 2011.

[20] Taeho Kim, Yeong-Tae Song, Lawrence Chung, and Dung T. Huynh. Software architecture analysis: a dynamic slicing approach. *ACIS International Journal of Computer & Information Science*, 1:91–103, March 2000.

[21] Bogdan Korel, Inderdeep Singh, Luay Tahat, and Boris Vaysburg. Slicing of state-based models. In *Proc. of the IEEE International Conference on Software Maintenance (ICSM'03)*, 2003.

[22] Sébastien Labbé and Jean-Pierre Gallois. Slicing communicating automata specifications: polynomial algorithms for model reduction. *Formal Aspects of Computing*, 20:563–595, 2008.

[23] Jaiprakash T. Lallchandani and R. Mall. A dynamic slicing technique for UML architectural models. *IEEE Transactions on Software Engineering*, 99, 2011.

[24] Kevin Lano and Shekoufeh Kolahdouz-Rahimi. Slicing of UML models using model transformations. In *International Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, pages 228–242. 2010.

[25] Arthorn Luangsodsai and Chris Fox. Concurrent statechart slicing. In *Second Conference on Computer Science and Electronic Engineering Conference (CEEC)*, pages 1–7, 2010.

[26] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of MODELS/UML'2005*, pages 264–278, 2005.

[27] Pierre-Alain Muller, Frédéric Fondement, Benoit Baudry, and Benoit Combemale. Modeling modeling modeling. *Software and Systems Modeling*, pages 1–13, 2010.

[28] Jim Q. Ning, Andre Engberts, and W. Voytek Kozaczynski. Automated support for legacy code understanding. *Commun. ACM*, 37(5):50–57, 1994.

[29] Juergen Rilling and Tuomas Klemola. Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, pages 115–124. IEEE Computer Society, 2003.

[30] Philip Samuel and Rajib Mall. Slicing-based test case generation from UML activity diagrams. *SIGSOFT Softw. Eng. Notes*, 34:1–14, 2009.

[31] Sagar Sen, Naouel Moha, Benoit Baudry, and Jean-Marc Jézéquel. Meta-model Pruning. In *12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, 2009.

[32] Asadullah Shaikh, Robert Clarisó, Uffe Kock Wiil, and Nasrullah Memon. Verification-driven slicing of UML/OCL models. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 185–194. ACM, 2010.

[33] Asadullah Shaikh, Uffe Kock Wiil, and Nasrullah Memon. UOST: UML/OCL aggressive slicing technique for efficient verification of models. In *System Analysis and Modeling: About Models - 6th International Workshop SAM'10*, pages 173–192, 2010.

[34] Asadullah Shaikh, Uffe Kock Wiil, and Nasrullah Memon. Evaluation of tools and slicing techniques for efficient verification of UML/OCL class diagrams. *Advances in Software Engineering*, vol. 2011:18 pages, 2011.

[35] Josep Silva. A vocabulary of program-slicing based techniques. *ACM Computing Surveys*, 2011.

[36] Jim Steel, Robin Drogemuller, and Bianca Toth. Model interoperability in building information modelling. *Software and Systems Modeling*, pages 1–11, 2010.

[37] Jim Steel and Jean-Marc Jézéquel. On model typing. *Journal of Software and Systems Modeling (SoSyM)*, 6(4):401–414, December 2007.

[38] Margaret-Anne D. Storey, F. David Fracchia, and Hausi A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, 44(3):171–185, 1999.

[39] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[40] Zoltán Ujhelyi, Ákos Horváth, and Dániel Varró. Towards dynamic backward slicing of model transformations. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 404–407. IEEE Computer Society, 2011.

[41] Sara Van Langenhove and Albert Hoogewijs. SVtL: system verification through logic tool support for verifying sliced hierarchical statecharts. In *Proceedings of the 18th international conference on Recent trends in algebraic development techniques*, pages 142–155, 2007.

[42] G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 107–119. ACM, 1991.

[43] Ji Wang, Wei Dong, and Zhichang Qi. Slicing hierarchical automata for model checking UML statecharts. In *Formal Methods and Software Engineering, 4th International Conference on Formal Engineering Methods, ICFEM 2002*, pages 435–446, 2002.

[44] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[45] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30:1–36, 2005.

[46] Nisansala Yatapanage, Kirsten Winter, and Saad Zafar. Slicing behavior tree models for verification. In *IFIP Advances in Information and Communication Technology*, pages 125–139, 2010.

[47] Jianjun Zhao. Applying slicing technique to software architectures. In *Fourth IEEE International Conference on Engineering of Complex Computer Systems, ICECCS'98*, pages 87 –98, 1998.