Software and Systems Modeling manuscript No.
(will be inserted by the editor)

A Generic Model Decomposition Technique and its Application to the Eclipse Modeling Framework

Qin Ma[1,2], Pierre Kelsen[1,2], Christian Glodt[1]

[1] FSTC, University of Luxembourg

[2] SnT, University of Luxembourg

Received: date / Revised version: date

Abstract Model-driven software development aims at easing the process of software development by using models as primary artifacts. Although less complex than the real systems they are based on, models tend to be complex nevertheless, thus making the task of handling them non-trivial in many cases. In this paper we propose a generic model decomposition technique to facilitate model management by decomposing complex models into smaller sub-models that conform to the same metamodel as the original model. The technique is based upon a formal foundation that consists of a formal capturing of the concepts of models, metamodels, and model conformance; a formal constraint language based on EssentialOCL; and a set of formally proved properties of the technique. We organize the decomposed sub-models in a mathematical structure as a lattice, and design a linear-time algorithm for constructing this decomposition.

The generic model decomposition technique is applied to the Eclipse Modeling Framework (EMF) and the result is used to build a solution to a specific model comprehension problem of Ecore models based upon model pruning. We report two case studies of the model comprehension method: one in BPMN and the other in fUML.

Key words MDE, EMF, Model decomposition, Model comprehension, Linear-time algorithm, Sub-model lattice, OCL, EssentialOCL, BPMN, fUML.

1 Introduction

In model-driven software development models are the primary artifacts. Typically several models are used to describe the different concerns of a system. One of the main motivations for using models is the problem of dealing with the complexity of real systems: because models

represent abstractions of a system, they are typically less complex than the systems they represent.

Nevertheless models for real systems can be complex themselves and thus may require aids for facilitating human comprehension. The problem of understanding complex models is at the heart of this paper. We propose to rely on a model decomposition technique that subdivides models into smaller relevant sub-models to aid in their comprehension.

An example of a concrete application scenario is the following: when trying to understand a large model, one starts with a subset of model elements that one is interested in (such as the concept of Class in the UML metamodel). Our method allows to construct a small sub-model of the initial model that contains all entities of interest and that conforms to the original metamodel (in the case of UML the original metamodel would be MOF). The latter condition ensures that the sub-model can be viewed in the same way as the original model and that it has a well-defined semantics. The smaller size (compared to the original model) should facilitate comprehension.

Instead of providing a particular solution to the specific comprehension problem, we first study a more general model decomposition problem in an abstract setting. More specifically, the decomposition problem deals with the following: given a metamodel and a model conforming to the metamodel, how does one derive all the conformant sub-models and how are the sub-models related to each other? Based upon a well established formal foundation that consists of a formal capturing of the concepts of models, metamodels, and model conformance, we propose a linear time algorithm to build the decomposition hierarchy of a model from which all the conformant sub-models can be constructed in a straightforward manner. We prove formally the correctness of the algorithm, and present the mathematical structure of these conformant sub-models as a lattice. A lattice (drawn upon order theory) is a partially ordered set in which any two elements have a least upper bound and a greatest lower bound. The original model is the greatest element in the lattice at the top and the empty sub-model is the least element in the lattice at the bottom.

The solution to the model decomposition problem is generic in two ways: first, the technique can be employed to decompose any model conforming to any metamodel; second, it considers the collection of relevant sub-models in its totality rather than a single sub-model.

The generic solution to the model decomposition problem is then customized to target a solution to the specific model comprehension problem we discussed earlier. This is achieved in two steps. First, we instantiate the decomposition solution to work within a concrete environment: Eclipse Modeling Framework (EMF) and EssentialOCL [27]. Second, the smallest sub-model in the decomposition hierarchy that contains all the concepts of interest is selected as an answer to the comprehension problem.

We implemented the model comprehension approach in an Ecore model comprehension tool [1] and carried out two case studies for validation. The first case study takes place in the context of BPMN [26] (business process model and notation), where we try to comprehend the *Gateway* concept. The second case study is about understanding the *Class* concept and *Namespace* concept in fUML [25] (describing a subset of executable UML models). In the first case, the size of the model for comprehension decreases by 93% (in terms of the number of model elements). In the second case, the size decreases by 62% and 89% (respectively for *Class* and *Namespace*).

Roadmap Metamodels and models are specified in practice with concrete tools. An abstraction layer is derived on top of all these concrete environments to retain only concepts that are relevant for model decomposition. We present the abstraction layer in Section 2 with formal definitions of models, metamodels and model conformance, and describe a technique for model decomposition in Section 3. We prove that the sub-models are conformant to the same metamodel as the original model and can be organized in a mathematical structure called the lattice of sub-models. Being an abstract metamodeling and modeling environment, the model decomposition technique defined at this level of abstraction is generally usable in any concrete environment. In this paper, we demonstrate its usage in the Eclipse Modeling Framework (EMF): Section 4 lays the ground for the usage of the model decomposition technique by providing a short summary of EMF; Section 5 presents a concrete yet formal constraint language, called CoreOCL (a core of EssentialOCL), for the specification of invariants attached to metamodels; and Section 6 discusses the actual steps involved, namely first establishing an alignment between EMF and the abstraction layer, then customizing the general model decomposition algorithm for EMF model decomposition, and finally discussing the soundness of the customized algorithm. We report the application of the model decomposition technique in a concrete scenario for Ecore model comprehension in Section 7 and present the results of two case studies. We evaluate our approach in Section 8 and discuss related work in Section 9. Finally, we present concluding remarks and point out some interesting directions for future work in the last section.

Extension statement We have presented a previous version of the model decomposition technique in [22] at the Fourteenth International Conference on Fundamental Approaches to Software Engineering (FASE 2011). Thanks to the valuable comments and suggestions from the anonymous referees of FASE 2011, we here are happy to present an extended and improved version of the model decomposition technique completed with all its formal treatments. Notably, three new pieces of work have been carried out in this extension, namely, a new formal constraint language called CoreOCL (reported in Section 5),

the usage of the model decomposition technique in the Eclipse Modeling Framework in general (reported in Section 4 and 6), and in the Ecore model comprehension example in particular (reported in Section 7.1).

2 Models and Metamodels

OMG puts forward MOF [24] as a platform independent framework for defining, manipulating and integrating metamodels and models. The MOF standard has been implemented in various concrete environments to support (meta-)modeling in real life. Examples of such concrete environments include EMF (the Eclipse Modeling Framework) [1], Kermeta (Kernel Metamodeling) [2], AMMS (ATLAS Model Management Architecture) [3], and MOFLON [4]. Inspired by MOF, we abstract from the metamodeling and modeling concepts present in mainstream concrete tools and formalize a set of definitions of models, metamodels, and model conformance, where only model decomposition relevant information is kept.

The following notational conventions will be used:

1. For any tuple p, we use $\mathsf{fst}(p)$ to denote its first element, $\mathsf{snd}(p)$ to denote its second element, and $\mathsf{trd}(p)$ to denote its third element.
2. For any set s, we use $\sharp s$ to denote its cardinality.

[1] http://www.eclipse.org/modeling/emf/
[2] http://www.kermeta.org/
[3] http://wiki.eclipse.org/AMMA
[4] http://www.moflon.org/

2.1 Metamodels

A metamodel defines (the abstract syntax of) a language for expressing models to be decomposed. A metamodel consists of the following parts: a finite set of metaclasses; a finite set of associations between metaclasses; and a finite inheritance relation between metaclasses. Moreover, a set of invariants may be specified in the contexts of metaclasses as additional well-formedness rules imposed upon models.

Definition 1 (Metamodel) *A metamodel is defined by a tuple* $\mathbb{M} = (\mathbb{N}, \mathbb{A}, \mathbb{H}, \mathbb{Inv}, \mathsf{s}, \mathsf{t}, \mu_{\mathsf{s}}, \mu_{\mathsf{t}}, \mathsf{ctx})$ *where:*

- $\mathbb{N}$ *is the set of metaclasses, and* $\mathbb{n} \in \mathbb{N}$ *ranges over it.*
- $\mathbb{A}$ *is the set of (directed) associations between metaclasses and* $\mathbb{a} \in \mathbb{A}$ *ranges over it.*
- $\mathbb{H} \subseteq \mathbb{N} \times \mathbb{N}$ *denotes the inheritance relation among metaclasses. The transitive closure of* $\mathbb{H}$ *should be irreflexive. Namely, a metaclass cannot inherit, directly or indirectly, from itself. The subtyping relation* ($\preccurlyeq$) *between metaclasses is defined as the reflexive and transitive closure of* $\mathbb{H}$.
- $\mathbb{Inv}$ *is the set of invariants and* $\mathbb{inv} \in \mathbb{Inv}$ *ranges over it.*
- $\mathsf{s} : \mathbb{A} \to \mathbb{N}$ *and* $\mathsf{t} : \mathbb{A} \to \mathbb{N}$ *are two functions from associations to metaclasses. They specify respectively the types of the source and target ends of an association.*
- $\mu_{\mathsf{s}} : \mathbb{A} \to \mathbb{\Sigma}$ *and* $\mu_{\mathsf{t}} : \mathbb{A} \to \mathbb{\Sigma}$ *are two functions from associations to well-formed multiplicities. They specify*

respectively the multiplicities of the source and target ends of an association. $\mathbb{\Sigma} \subseteq \mathsf{Nat} \times \{\mathsf{Int}^+ \cup \{*\}\}$ *captures the set of multiplicities, where* Nat *is the set of natural numbers (i.e. non-negative integers) and* Int^+ *is the set of positive integers. Multiplicities are ranged over by* $\sigma \in \mathbb{\Sigma}$*. We assume that all the multiplicities in* $\mathbb{\Sigma}$ *are well-formed, namely* $\mathsf{snd}(\sigma) = *$ *or* $\mathsf{fst}(\sigma) \leq \mathsf{snd}(\sigma)$*. A well-formed multiplicity defines an inclusive interval from a lower bound to an upper bound. An asterisk* $*$ *is used for denoting an unlimited upper bound.*

– $\mathsf{ctx} : \mathbb{Inv} \to \mathbb{N}$ *is a function from invariants to metaclasses. It specifies the context metaclass of an invariant.*

In addition to the context, an invariant also comes with a body which, in practice, is either documented in natural language, or specified as expressions of a constraint language such as EssentialOCL [27], or implemented in terms of code in a programing language such as Java.

Note that we omit the following concepts in metamodels on purpose:

– *Attributes* are treated as a special kind of associations, where one end of the connected metaclasses is a data type. Consequently, data types (including both primitive types e.g., integers and booleans and enumeration types) are all metaclasses.

– *Composition* or *containment* relations are treated as a special kind of associations with extra constraints:
  1. the source multiplicity of a composition association is (0..1);
  2. a metaclass instance cannot be the target of more than one composition link (i.e., instantiations of composition associations);
  3. and a metaclass instance cannot be connected to itself via a sequence of composition links.

Moreover, we have also omitted the concept of operations in metaclasses because differently from associations (or attributes or references), operations only exist at the level of the metamodel. Side effect free helper operations that are used for invariant specifications can be defined directly with the corresponding invariants when needed.

2.2 Models

A model is built by instantiating the metaclasses and associations of the metamodel.

Definition 2 (Model) *A model is defined by a tuple* $\mathsf{M} = (\mathbb{M}, \mathsf{N}, \mathsf{A}, \tau, \mathsf{src}, \mathsf{tgt})$ *where:*

– $\mathbb{M}$ *is the metamodel in which the model is expressed.*

– N *is the set of metaclass instantiations, and* $\mathsf{n} \in \mathsf{N}$ *ranges over it. They are often referred to as instances.*

– A *is the set of association instantiations, and* $\mathsf{a} \in \mathsf{A}$ *ranges over it. They are often referred to as links.*

– $\tau : (\mathsf{N} \to \mathbb{N}) \cup (\mathsf{A} \to \mathbb{A})$ *is the typing function. It records the type information of the instances and links in the model, i.e., from which metaclasses or associations of the metamodel* $\mathbb{M}$ *they are instantiated.*

– $\mathsf{src} : \mathsf{A} \to \mathsf{N}$ *and* $\mathsf{tgt} : \mathsf{A} \to \mathsf{N}$ *are two functions from links to instances. They specify respectively the source and target ends of a link.*

2.3 Model Conformance

Not all models following the definition above are valid, or "conform to" the metamodel. A valid model should satisfy all the typing, multiplicity, and extra well-formedness invariants captured in the metamodel.

Definition 3 (Model conformance) *We say a model* $\mathsf{M} = (\mathbb{M}, \mathsf{N}, \mathsf{A}, \tau, \mathsf{src}, \mathsf{tgt})$ *conforms to its metamodel* $\mathbb{M}$ *or is valid when the following conditions are met:*

1. *Typing condition: links only connect instances whose types are compatible with (i.e., subtypes of) the metaclasses specified for the corresponding association ends. Namely,* $\forall \mathsf{a} \in \mathsf{A}$*, we have both* $\tau(\mathsf{src}(\mathsf{a})) \preccurlyeq \mathsf{s}(\tau(\mathsf{a}))$ *and* $\tau(\mathsf{tgt}(\mathsf{a})) \preccurlyeq \mathsf{t}(\tau(\mathsf{a}))$.
2. *Multiplicity condition: the numbers of links must fall in the ranges specified by the multiplicities of the corresponding associations. We shall consider both the source and target multiplicities.*
   (a) $\forall \mathsf{n} \in \mathsf{N}, \mathbb{a} \in \mathbb{A}$*, if* $\tau(\mathsf{n}) \preccurlyeq \mathsf{t}(\mathbb{a})$*, then let* $k = \sharp\{\mathsf{a} \in \mathsf{A} \mid \tau(\mathsf{a}) = \mathbb{a} \text{ and } \mathsf{tgt}(\mathsf{a}) = \mathsf{n}\}$ *(i.e., the number of* $\mathbb{a}$*-typed links ending at the instance* n *in model* M*), we must have* $k \geq \mathsf{fst}(\mu_{\mathsf{s}}(\mathbb{a}))$ *(i.e., the lower bound of the source multiplicity of* $\mathbb{a}$*) and* $k \leq \mathsf{snd}(\mu_{\mathsf{s}}(\mathbb{a}))$ *(i.e., the upper bound of the source multiplicity of* $\mathbb{a}$*) in case the latter is not* $*$*;*
   (b) $\forall \mathsf{n} \in \mathsf{N}, \mathbb{a} \in \mathbb{A}$*, if* $\tau(\mathsf{n}) \preccurlyeq \mathsf{s}(\mathbb{a})$*, then let* $k = \sharp\{\mathsf{a} \in \mathsf{A} \mid \tau(\mathsf{a}) = \mathbb{a} \text{ and } \mathsf{src}(\mathsf{a}) = \mathsf{n}\}$ *(i.e., the number of* $\mathbb{a}$*-typed links leaving the instance* n *in model* M*), we must have* $k \geq \mathsf{fst}(\mu_{\mathsf{t}}(\mathbb{a}))$ *(i.e., the lower bound of the target multiplicity of* $\mathbb{a}$*) and* $k \leq \mathsf{snd}(\mu_{\mathsf{t}}(\mathbb{a}))$ *(i.e., the upper bound of the target multiplicity of* $\mathbb{a}$*) in case the latter is not* $*$*.*
3. *Invariant condition: all invariants should hold.* $\forall \mathbb{inv} \in \mathbb{Inv}$*,* $\forall \mathsf{n} \in \mathsf{N}$ *where* $\tau(\mathsf{n}) \preccurlyeq \mathsf{ctx}(\mathbb{inv})$*, (i.e., the type of* n *is compatible with the context metaclass,) the invariant* $\mathbb{inv}$ *should evaluate to* true *in model* M *for the contextual instance* n*. (Section 5 defines the evaluation semantics of invariants written in CoreOCL.)*

3 Model Decomposition

3.1 Criteria

Model decomposition starts from a model that conforms to a metamodel, and decomposes it into smaller parts. Our model decomposition technique is designed using the following as main criterion: the derived parts should be valid models conforming to the original metamodel. Achieving this goal has two main advantages:

1. the derived parts, being themselves valid models, can be comprehended on their own according to the fa-

miliar abstract syntax and semantics (if defined) of the modeling language;

2. the derived parts can be wrapped up into modules and reused in the construction of other system models, following a modular model composition paradigm such as [21].

In general, a decomposed smaller part of a model M is a sub-model whose instance set is a subset of that of M, and whose link set is a subset of the links of M restricted to the instance subset. In our model decomposition technique we consider a particular kind of sub-models, called *instance induced sub-models*, as we view models as essentially sets of instances, augmented with links among these instances. As a consequence, induced sub-models include all the links involving the instances included in the sub-models. In graph-theoretic terms this corresponds to induced subgraphs [12] (if we view models as graphs).

Definition 4 (Instance induced sub-model) *We say a model* $\mathsf{M}' = (\mathbb{M}, \mathsf{N}', \mathsf{A}', \tau', \mathsf{src}', \mathsf{tgt}')$ *is an instance induced sub-model of another model* $\mathsf{M} = (\mathbb{M}, \mathsf{N}, \mathsf{A}, \tau, \mathsf{src}, \mathsf{tgt})$ *if and only if:*

1. $\mathsf{N}' \subseteq \mathsf{N}$;
2. $\mathsf{A}' = \{\mathsf{a} \mid \mathsf{a} \in \mathsf{A}$ *and* $\mathsf{src}(\mathsf{a}) \in \mathsf{N}'$ *and* $\mathsf{tgt}(\mathsf{a}) \in \mathsf{N}'\}$;
3. τ' *is the restriction of* τ *to* N' *and* src' *and* tgt' *are the restrictions of* src *and* tgt *to* A'.

From now on, all sub-models we talk about are instance induced unless mentioned otherwise explicitly. Namely, when constructing a sub-model, we shall only discuss the inclusion of instances and let the inclusion of links be induced by the instances included in the sub-model.

In order to make the sub-model M' also conform to $\mathbb{M}$, we will propose three conditions - one for the metamodel (Condition 2 below, regarding the nature of the invariants) and two conditions for the sub-model (Conditions 1 and 3). Altogether these three conditions will be sufficient to ensure conformance of the sub-model.

The starting point of our investigation is the definition of conformance (Definition 3). Three conditions must be met in order for sub-model M' to conform to metamodel $\mathbb{M}$.

We first tackle the most sophisticated one: the invariant condition in the conformance definition. It requires that all metamodel invariants are satisfied in sub-model M'. These invariants are known to be satisfied in model M because M conforms to the metamodel. Therefore, it is sufficient to maintain the same evaluation of these invariants in M'.

In order to achieve this goal, let us first introduce some knowledge about invariant evaluation in models. Invariants are evaluated on so called well-formed evaluation points defined as follows:

Definition 5 (Well-formed evaluation point) *We call a triplet of the following form* $(\mathbb{inv}, \mathsf{M}, \mathsf{n})$ *an evaluation point of an invariant* $\mathbb{inv}$ *in a model* M *on an instance* n. *An evaluation point is well-formed if* $\mathbb{inv}$ *is*

defined for the metamodel of M*,* n *is an instance of* M*, and the type of* n *is a subtype of the context metaclass of* $\mathbb{inv}$.

The result of evaluating a well-formed evaluation point is determined by its *scope*, defined below:

Definition 6 (Scope of invariant evaluation) *The scope of a well-formed evaluation point* $(\mathbb{inv}, \mathsf{M}, \mathsf{n})$ *is a sub-model of* M *induced by all the instances that are referenced during the evaluation of the invariant* $\mathbb{inv}$ *in model* M *on the contextual instance* n.

For example, CoreOCL provides three ways to reference instances in invariants:

1. referencing the contextual instance via keyword self;
2. following links to reference target instances;
3. using the $\mathsf{AllInstances}$ operation to reference all the instances of a given metaclass.

Section 5 discusses how scopes are constructed along the evaluation of invariants. Moreover, a formal proof is also presented in Section 5 to demonstrate that letting a sub-model contain the scopes of invariant evaluations suffices to preserve the invariants holding in the original model (see Theorem 5). More specifically, given an evaluation point $(\mathbb{inv}, \mathsf{M}, \mathsf{n})$ and a sub-model M' of M, letting M' include all the instances of the scope of $(\mathbb{inv}, \mathsf{M}, \mathsf{n})$ suffices to have the same result for evaluating both $(\mathbb{inv}, \mathsf{M}, \mathsf{n})$ and $(\mathbb{inv}, \mathsf{M}', \mathsf{n})$, i.e., the invariant $\mathbb{inv}$ has the same evaluation in both M and M'.

In theory, a metamodel can specify an invariant, where an evaluation point of this invariant in a model on a contextual instance has a scope that spans the whole instance set of the model. Following the discussion above, if the model decomposition technique is expected to always preserve the evaluation of such "global" invariants in a sub-model that contains the contextual instance, the sub-model should effectively include all the instances of the original model (i.e., equivalent to the original model), hence leaves no room for effective sub-modeling.

Fortunately, most of the scopes implied by invariants in practice involves only a portion of the original model that is reachable from the contextual instance. We refer to such kind of invariants as *forward invariants*, precisely defined below:

Definition 7 (Reachability) *Given a model* M*, we say an instance* $\mathsf{n}' \in \mathsf{N}$ *is reachable from an instance* $\mathsf{n} \in \mathsf{N}$*, if and only if there exists* $\mathsf{a}_1, \mathsf{a}_2, \ldots, \mathsf{a}_k$ *in* A *such that* $\mathsf{src}(\mathsf{a}_1) = \mathsf{n}$*,* $\mathsf{tgt}(\mathsf{a}_i) = \mathsf{src}(\mathsf{a}_{i+1})$*, for all* i*,* $1 \leq i \leq k-1$*, and* $\mathsf{tgt}(a_k) = \mathsf{n}'$.

Definition 8 (Forward invariant) *An invariant* $\mathbb{inv}$ *is forward if for any well-formed evaluation point* $(\mathbb{inv}$*,* M*,* $\mathsf{n})$*, all the instances in the corresponding scope* N_S *are reachable from* n.

Following the discussion above, we impose a condition on the sub-model to include all the instances that are reachable from an instance that is already included, formally expressed below:

Condition 1 $\forall \mathsf{a} \in \mathsf{A}$, $\mathsf{src}(\mathsf{a}) \in \mathsf{N}'$ *implies* $\mathsf{tgt}(\mathsf{a}) \in \mathsf{N}'$.

Meanwhile, we only allow forward invariants, as expressed in the following condition over the metamodel:

Condition 2 *All invariants of metamodel* $\mathbb{M}$ *are forward invariants.*

It is not difficult to see that Conditions 1 and 2 together guarantee that an invariant satisfied on contextual instance n in M is also satisfied in the instance induced sub-model M' on the same contextual instance, since the evaluations of the two points are indeed determined by the same scope (and all sub-models are instance induced). Consequently, the invariant condition in the conformance definition is ensured.

The multiplicity condition for conformance concerns the numbers of links in models. The number of links of a given association may decrease from the original model M to the sub-model M' in cases where an instance connected at an end of a link is not included in M'. We need to examine both the number of links ending at the instance in M', which must agree with the source multiplicity of the corresponding association in the metamodel, and the number of links leaving the instance in M', which must agree with the target multiplicity of the corresponding association.

The multiplicity condition for links leaving an instance of the sub-model M' is ensured by Condition 1 and the fact that M' is instance induced. The two together guarantee that the number of links leaving an instance in M' is exactly the same as the number in M, hence remains within the range allowed by the target multiplicity.

To ensure the multiplicity condition for links ending at an instance of the sub-model, we will introduce the notion of *fragmentable links*, whose type (i.e., the corresponding association) has an unconstrained (i.e., being 0) lower bound for the source multiplicity.

Definition 9 (Fragmentable link) *Given a model* $\mathsf{M} = (\mathbb{M}, \mathsf{N}, \mathsf{A}, \tau, \mathsf{src}, \mathsf{tgt})$*, a link* $\mathsf{a} \in \mathsf{A}$ *is fragmentable if the lower bound of the source multiplicity of the corresponding association is* 0*, i.e.,* $\mu_{\mathsf{s}}(\tau(\mathsf{a})) = (0, \_)$*, where* $\_$ *represents an arbitrary upper bound.*

Fragmentable incoming links (together with the source instances) of M to instances in M' are safe to exclude but this is not the case for non-fragmentable links, which should all be included. We thus obtain the following condition on sub-model M':

Condition 3 $\forall \mathsf{a} \in \mathsf{A}$ *where* a *is non-fragmentable,* $\mathsf{tgt}(\mathsf{a}) \in \mathsf{N}'$ *implies* $\mathsf{src}(\mathsf{a}) \in \mathsf{N}'$.

Finally, the typing condition for conformance follows directly from the fact that M' is an instance induced sub-model of M and M conforms to $\mathbb{M}$.

Summarizing the discussion above, we thus obtain the following result:

Theorem 1 *Given a metamodel* $\mathbb{M}$*, a model* M*, and an instance induced sub-model* M' *of* M*, suppose that:*

1. *model* M *conforms to* $\mathbb{M}$*;*

2. model M' *satisfies Conditions 1 and 3;*

3. metamodel $\mathbb{M}$ *satisfies Condition 2;*

then model M' *also conforms to the metamodel* $\mathbb{M}$*.*

Proof The result follows from the discussion above. □

3.2 Algorithm

From hereon we shall assume that the metamodel under consideration satisfies Condition 2. In this subsection we describe an algorithm that finds, for a given model M, a partition of M such that any sub-model of M that satisfies both Conditions 1 and 3 can be derived from the partition by uniting some components of the partition.

We reach the goal in two steps: (1) ensure Condition 1 with respect to only non-fragmentable links and Condition 3; (2) ensure Condition 1 with respect to fragmentable links. Details of each step are discussed below.

Treating instances as nodes and links as edges, models are just graphs. For illustration purpose, consider an example model as presented in Figure 1 where all fragmentable links are indicated by two short parallel lines crossing the links.

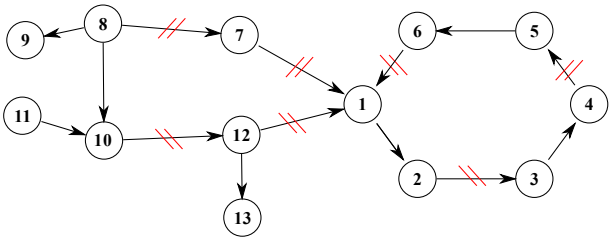

Fig. 1 An example model

Let G be the graph derived by removing the fragmentable links from M. Because all the links in G are non-fragmentable, for a sub-graph of G to satisfy both Conditions 1 and 3, an instance is included in the sub-graph if and only if all its ancestor and descendant instances are also included. An instance together with its ancestors and descendants, from the point of view of graph theory, constitute a weakly connected component (wcc) of graph G (i.e., a connected component if we ignore edge directions). The first step of the model decomposition computes all such wcc's of G, which disjointly cover all the instances in model M, then puts back the fragmentable links. We collapse all the instances that belong to one wcc into one node, (referred to as *a wcc-node* in contrast to the original nodes), and refer to the result as graph W. After the first step, the corresponding graph W of the example model contains six wcc-nodes inter-connected by fragmentable links. We show W in Figure 2, where the instances grouped in each wcc-node remain visible (in dashed border style) for traceability purpose.

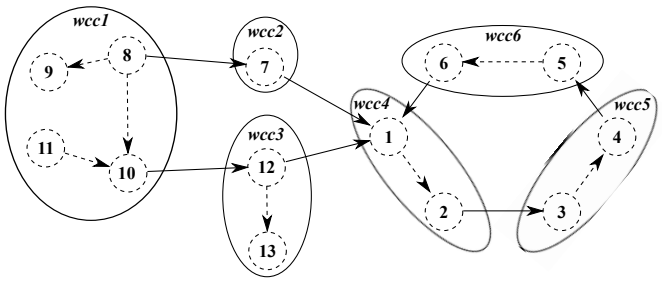

Fig. 2 The corresponding W graph of the example model after step 1

A sub-model of M that is induced by the instances grouped in one wcc-node in W satisfies Condition 1 and Condition 3, but only with respect to non-fragmentable links for Condition 1, because wcc's are computed in the context of G where fragmentable links are removed. The second step of the model decomposition starts from graph W and tries to satisfy Condition 1 with respect to fragmentable links, i.e., following outgoing fragmentable links. More specifically, we compute all the strongly connected components (scc's) in W (see [33] for a definition of strongly connected components) and collapse all the wcc-nodes that belong to one scc into one node, (referred to as *an scc-node*), and refer to the result as graph D. After the second step, the corresponding graph D of the example model looks as depicted in Figure 3, where the original model instances and previous wcc-nodes that are grouped in each scc-node are also shown (in dashed border style) for traceability purpose. The three wcc-nodes

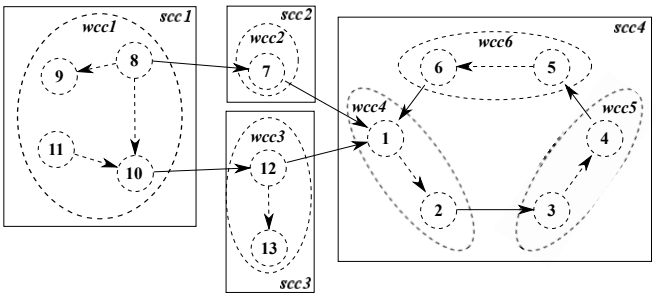

Fig. 3 The corresponding D graph of the example model after step 2

*wcc*4, *wcc*5 and *wcc*6 of graph W are collapsed into one scc-node *scc*4 because they lie on a (directed) cycle.

Note that we only collapse wcc-nodes of a strongly connected component in the second step instead of any reachable ones following outgoing fragmentable links in W, because we do not want to lose any potential sub-model of M satisfying both Conditions 1 and 3 on the way. More precisely, a set of instances is grouped in an scc-node only if for every instance induced sub-model M′ of M satisfying both Conditions 1 and 3, it is either completely contained in M′ or disjoint with M′, i.e., no such M′ can tell the instances in the set apart.

The computational complexity of the above algorithm is dominated by the complexity of computing weakly and strongly connected components in the model graph. Computing weakly connected components amounts to computing connected components if we ignore the direction of the edges. We can compute connected components and strongly connected components in linear time using depth-first search [33]. Thus the overall complexity is linear in the size of the model graph.

3.3 Correctness

Graph D obtained at the end of the algorithm is a DAG (Directed Acyclic Graph) with all the edges being fragmentable links. Graph D represents a partition of the original model M where all the instances that are grouped in an scc-node in D constitute a component in the partition. We call graph D *the decomposition hierarchy* of model M.

To relate the decomposition hierarchy to the sub-models, we introduce the concept of an *antichain-node*. An antichain-node is derived by collapsing a (possibly empty) antichain of scc-nodes (i.e., a set of scc-nodes that are neither descendants nor ancestors of one another, the concept of antichain being borrowed from order theory) plus their descendants (briefly an antichain plus descendants) in the decomposition hierarchy. For example, in the decomposition hierarchy of the example model given in Figure 3, the two scc-nodes: *scc*2 and *scc*3 constitute an antichain, and collapsing them with their descendant *scc*4, gives rise to an antichain-node, which groups the nine instances that are previously grouped in the collapsed scc-nodes. Sub-models are then induced by the instances grouped in antichain-nodes.

To demonstrate the correctness of the algorithm, we prove both the soundness, i.e., a sub-model induced by the instances grouped in an antichain-node satisfies both Conditions 1 and 3, and the completeness, i.e., any sub-model satisfying both Conditions 1 and 3 can be induced by the instances grouped in an antichain-node. We formally capture the correctness by the following theorem:

Theorem 2 *Given a model* $\mathsf{M} = (\mathbb{M}, \mathsf{N}, \mathsf{A}, \tau, \mathsf{src}, \mathsf{tgt})$ *and an instance induced sub-model* $\mathsf{M}' = (\mathbb{M}, \mathsf{N}', \mathsf{A}', \tau', \mathsf{src}', \mathsf{tgt}')$ *of* M, M' *satisfies both Conditions 1 and 3 if and only if there exists a corresponding antichain-node of the decomposition hierarchy of* M *where* M' *is induced by all the instances grouped in this antichain-node.*

Proof See Appendix A.1. □

3.4 The lattice of sub-models

Recall that a lattice is a partially-ordered set in which every pair of elements has a least upper bound and a greatest lower bound. Thanks to Theorem 2, we can now refer to an instance induced sub-model M' of model M that satisfies both Conditions 1 and 3 by the corresponding antichain-node α of the decomposition hierarchy of M. Given a model M, all the instance induced sub-models that satisfy both Conditions 1 and 3 constitute a lattice ordered by the relation "is a sub-model of", referred to as *the sub-model lattice* of M. Let α_1 and α_2 denote two such sub-models. The least upper bound $(\alpha_1 \vee \alpha_2)$ and the greatest lower bound $(\alpha_1 \wedge \alpha_2)$ are computed in the following way:

- $\alpha_1 \vee \alpha_2$ collapses all the scc-nodes that are collapsed in either α_1 or α_2. If an scc-node is collapsed in either α_1 and α_2, so are all its descendants because α_1 and α_2 are antichain-nodes. Therefore, $\alpha_1 \vee \alpha_2$ is an antichain-node, i.e., an instance induced sub-model of M. Moreover, it is the least one of which both α_1 and α_2 are sub-models.
- $\alpha_1 \wedge \alpha_2$ collapses all the scc-nodes that are both collapsed in α_1 and α_2. If an scc-node is collapsed in both α_1 and α_2, so are all its descendants because α_1 and α_2 are antichain-nodes. Therefore, $\alpha_1 \wedge \alpha_2$ is an antichain-node, i.e., an instance induced sub-

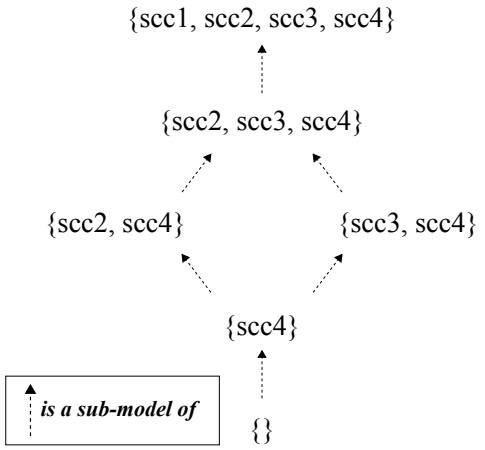

Fig. 4 The sub-model lattice of the example model in Figure 1 whose decomposition hierarchy is given in Figure 2

model of M. Moreover, it is the greatest one that is a sub-model of both α_1 and α_2.

The top of the sub-model lattice is M itself (whose corresponding antichain-node collapses all the scc-nodes), and the bottom is the empty sub-model (whose corresponding antichain-node collapses none scc-node).

For the example model discussed in Section 3.2 whose decomposition hierarchy is given in Figure 3, six possible antichain-nodes can be derived from the decomposition hierarchy, denoted by the set of scc-nodes that are collapsed. They are ordered in a lattice as shown in Figure 4.

3.5 Implementation

We have implemented the model decomposition technique [1] based on linear graph algorithms to compute connected components and strongly connected components [33]. The implementation takes a model of any metamodel that follows Definition 1 as input, and computes the decomposition hierarchy of it from which the sub-model lattice can be constructed by enumerating all the antichain-nodes of the decomposition hierarchy. Note that in the worst case where the decomposition hierarchy contains no edges, the size of the sub-model lattice equals the size of the power-set of the decomposition hierarchy, which is exponential. However, this does not impact the complexity of the implementation, which is linear in the size of the model graph, because we do not require an explicit representation of the sub-model lattice. Instead, the lattice and the sub-models in it are all derived from the decomposition hierarchy.

4 EMF in A Nutshell

The definitions of Section 2 capture a minimal abstraction of the metamodeling and modeling concepts that are relevant for model decomposition and common to mainstream tools. As a consequence, the model decomposition technique defined for this abstraction layer can be used to decompose models conforming to metamodels defined in any of these concrete environments. We demonstrate in this paper the usage of the model decomposition technique in EMF (the Eclipse Modeling Framework) — arguably the most popular modeling framework used by the model-driven software development community.

EMF [32] is a framework for metamodel specification and provides a reflective editor for model creation. A

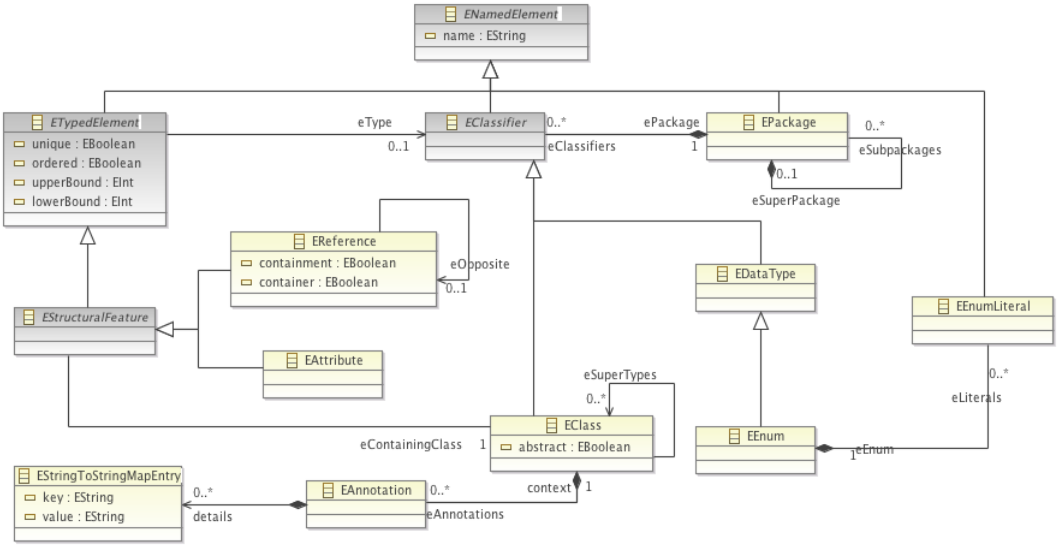

Fig. 5 Class diagram of a core part of Ecore focusing on metamodel specification and oriented to model decomposition.

metamodel in EMF is instantiated from Ecore (Ecore.ecore) — a meta metamodel aligned with the OMG EMOF [24] standard, and is accompanied with a constraint model that captures extra well-formedness invariants. This section lays the ground for the usage of our model decomposition technique in EMF by providing a short summary of metamodels, models and conformance in EMF. It by no means attempts to give a formal capture of EMF neither of Ecore. Works on this topic deserve their own contributions, such as [6] and [3].

The focus is put on the metamodeling capability of EMF and the applicability of the model decomposition technique in EMF. More specifically, the class diagram presented in Figure 5 represents a core part of Ecore that is oriented to this focus in the following sense:

1. Implementation relevant elements and generics are omitted.
2. The notion of operations is left out of metamodels. Side effect free helper operations that are used for constraint specifications will be defined directly in the accompanying constraint model when needed.

4.1 Metamodels in EMF

A metamodel $\mathbb{EM}$ in EMF (represented by a .ecore file) is constructed by instantiating the class diagram given in Figure 5. Briefly speaking, an EMF metamodel consists of sets of elements instantiated from the non-abstract classes (boxes with yellow background) in the diagram, and these elements are connected by instantiating the relations (lines) in the diagram, where all the conditions posed upon the relations with adornments (such as a black diamond) and multiplicities on the corresponding line ends must be respected. We summarize the main sets of elements of an EMF metamodel by a tuple $\mathbb{EM}$

$= (\mathbb{C}, \mathbb{DT}, \mathsf{DV}, \mathbb{Rf}, \mathbb{At})$, and explain the sets and their relations below.

1. A finite set of classes $\mathbb{C}$ ranged over by $\mathfrak{c} \in \mathbb{C}$ (instances of EClass).
   - A class $\mathfrak{c}$ may be abstract, i.e., $\mathfrak{c}$.abstract=true.
   - We can connect a class to another class within the relation eSuperTypes to say that the former inherits from the latter. The transitive closure of eSuperTypes should be irreflexive. We define the subtyping relation ($\preccurlyeq$) between classes as the reflexive and transitive closure of eSuperTypes.
   - We can associate an annotation (instance of EAnnotation) to a class with a specific key equal to ``constraints'' so as to enumerate the names of the invariants that are imposed on the class in the value of the annotation. Enumerated invariants are to be defined in the accompanying constraint model of the metamodel, with exactly the same context classes and invariant names.
2. A finite set of data types $\mathbb{DT}$ ranged over by $\mathfrak{dt} \in \mathbb{DT}$ (instances of EDataType and EEnum) to capture both primitive types, i.e., Integer, Boolean, String, and Real, predefined for any metamodel, and enumeration types, specific to the metamodel.
   - The subtyping relation ($\preccurlyeq$) between primitive types follows the conventional definition, i.e., Integer is a subtype of Real. No subtyping relation holds between enumeration types.
3. A finite set of data values DV ranged over by $\mathsf{dv} \in \mathsf{DV}$.
   - Values of metamodel specific enumeration types are explicitly specified by enumeration literals (instances of EEnumLiteral) to extensionally define the types.
   - Values of predefined primitive types are implicitly given following the conventional definitions.
   - A typing function from data values to data types: $\tau_{\mathsf{d}} : \mathsf{DV} \to \mathbb{DT}$ tells the type of a data value. Note that if the data value dv is an enumeration literal, $\tau_{\mathsf{d}}(\mathsf{dv})$ should coincide with dv.eEnum, i.e., dv is connected to $\tau_{\mathsf{d}}(\mathsf{dv})$ within the relation eEnum.
4. A finite set of references $\mathbb{Rf}$ ranged over by $\mathfrak{rf} \in \mathbb{Rf}$ (instances of EReference). A reference $\mathfrak{rf}$ has an owning class (aka. the source of the reference), specified by $\mathfrak{rf}$.eContainingClass.
5. A finite set of attributes $\mathbb{At}$ ranged over by $\mathfrak{at} \in \mathbb{At}$ (instances of EAttribute). An attribute $\mathfrak{at}$ has an owning class (aka. the source of the attribute), specified by $\mathfrak{at}$.eContainingClass.

References and attributes are both known as the structural features of a class. A structural feature, written $\mathfrak{sf}$, has itself the following features.

- A type (aka. the target of the structural feature), specified by $\mathfrak{sf}$.eType. Note that the type of a reference must be a class and the type of an attribute must be a data type.

- A multiplicity, specified by $\mathfrak{sf}.\mathsf{lowerBound}$ for its lower bound, which is a natural number (i.e., zero or positive integer), and $\mathfrak{sf}.\mathsf{upperBound}$ for its upper bound, which is either a positive integer or -1. Note that EMF implements an unlimited upper bound, i.e., $*$, by -1. In case the upper bound of the multiplicity is not -1, it must be greater than or equal to the lower bound.
- When $\mathfrak{sf}.\mathsf{ordered} = \mathsf{true}$, we call the structural feature $\mathfrak{sf}$ ordered.
- When $\mathfrak{sf}.\mathsf{unique} = \mathsf{true}$, we call the structural feature unique.

In addition to the common features above, references have the following extra features.

- A reference $\mathfrak{rf}$ is a containment, when $\mathfrak{rf}.\mathsf{containment} = \mathsf{true}$ (depicted by a line with a filled diamond at the source end in class diagrams).
- A reference $\mathfrak{rf}$ can have an opposite reference, specified by $\mathfrak{rf}.\mathsf{eOpposite}$, where the relation $\mathsf{eOpposite}$ should be functional, symmetric, and irreflexive.

The accompanying constraint model of a metamodel in EMF consists of a finite set of invariants $\mathbb{Einv}$, ranged over by $\mathfrak{einv}$, to provide specifications to the invariants enumerated in the metamodel. The context of an invariant $\mathfrak{einv}$ is the same as the class to which the corresponding annotation (i.e., the annotation in the value of which the name of the invariant is enumerated) is associated. The body of a invariant is given by an expression. We keep the notion of expression abstract to accommodate all potential languages for expression specification. In Section 5, we define a formal constraint language called CoreOCL for this purpose. In addition, a finite set of helper operations can be defined in the constraint model.

4.2 Models in EMF

A model EM in EMF is constructed by instantiating the classes of an EMF metamodel $\mathbb{EM}$ and assigning values to attributes and references owned by the classes. We summarize the components of a model in EMF by a tuple $\mathsf{EM} = (\mathbb{EM}, \mathsf{O}, \tau_{\mathsf{o}}, \mathsf{RA}, \tau_{\mathsf{ra}}, \mathsf{ra_t}, \mathsf{ra_s}, \mathsf{AA}, \tau_{\mathsf{aa}}, \mathsf{aa_s}, \mathsf{aa_t})$ and explain below.

1. A finite set of objects O ranged over by o.
2. An object typing function $\tau_{\mathsf{o}} : \mathsf{O} \to \mathbb{C}$ from objects to classes, to tell the type of an object, i.e., from which class the object is instantiated.
3. A finite set of reference assignments RA ranged over by ra.
4. Reference assignments involve three functions: $\tau_{\mathsf{ra}} : \mathsf{RA} \to \mathbb{Rf}$, $\mathsf{ra_s} : \mathsf{RA} \to \mathsf{O}$, and $\mathsf{ra_t} : \mathsf{RA} \to \mathsf{O}$ that specify respectively the type, the source, and the target of a reference assignment. The semantics of a reference assignment is $\mathsf{ra_s}(\mathsf{ra}).\tau_{\mathsf{ra}}(\mathsf{ra}) := \mathsf{ra_t}(\mathsf{ra})$.
5. The set of reference assignments RA is partially ordered, where for each ordered reference $\mathfrak{rf}$ of the metamodel $\mathbb{EM}$ and an object o of the model EM, $\{\mathsf{ra} \mid \mathsf{ra} \in \mathsf{RA} \text{ and } \tau_{\mathsf{ra}}(\mathsf{ra}) = \mathfrak{rf} \text{ and } \mathsf{ra_s}(\mathsf{ra}) = \mathsf{o}\}$ con-

stitutes a totally ordered subset of RA, (i.e., a chain in the partial order on RA).

6. A finite set of attribute assignments AA ranged over by aa.
7. Attribute assignments involve three functions: $\tau_{\mathsf{aa}} : \mathsf{AA} \to \mathbb{At}$, $\mathsf{aa_s} : \mathsf{AA} \to \mathsf{O}$, and $\mathsf{aa_t} : \mathsf{AA} \to \mathsf{DV}$ that specify respectively the type, the source, and the target of an attribute assignment. The semantics of an attribute assignment is $\mathsf{aa_s}(\mathsf{aa}).\tau_{\mathsf{aa}}(\mathsf{aa}) := \mathsf{aa_t}(\mathsf{aa})$.
8. The set of attribute assignments AA is partially ordered, where for each ordered attribute $\mathbb{at}$ of the metamodel $\mathbb{EM}$ and an object o of the model EM, $\{\mathsf{aa} \mid \mathsf{aa} \in \mathsf{AA} \text{ and } \tau_{\mathsf{aa}}(\mathsf{aa}) = \mathbb{at} \text{ and } \mathsf{aa_s}(\mathsf{aa}) = \mathsf{o}\}$ constitutes a totally ordered subset of AA, (i.e., a chain in the partial order on AA).

4.3 Conformance in EMF

In EMF, the conformance of a model EM to a metamodel $\mathbb{EM}$ requires the following conditions to hold:

1. Typing condition: all reference assignments are valid. For a reference assignment ra to be valid,
   (a) $\tau_{\mathsf{o}}(\mathsf{ra_s}(\mathsf{ra}))$ is a subtype of $\tau_{\mathsf{ra}}(\mathsf{ra}).\mathsf{eContainingClass}$, i.e., $\tau_{\mathsf{ra}}(\mathsf{ra})$ should be a reference defined for the class of $\mathsf{ra_s}(\mathsf{ra})$;
   (b) the class of $\mathsf{ra_t}(\mathsf{ra})$ is compatible with the type of $\tau_{\mathsf{ra}}(\mathsf{ra})$, i.e., $\tau_{\mathsf{o}}(\mathsf{ra_t}(\mathsf{ra}))$ is a subtype of $\tau_{\mathsf{ra}}(\mathsf{ra}).\mathsf{eType}$.
2. Attribute typing condition: all attribute assignments are valid. For an attribute assignment aa to be valid,
   (a) $\tau_{\mathsf{o}}(\mathsf{aa_s}(\mathsf{aa}))$ is a subtype of $\tau_{\mathsf{aa}}(\mathsf{aa}).\mathsf{eContainingClass}$, i.e., $\tau_{\mathsf{aa}}(\mathsf{aa})$ should be an attribute defined for the class of $\mathsf{aa_s}(\mathsf{aa})$;
   (b) the type of $\mathsf{aa_t}(\mathsf{aa})$ is compatible with the type of $\tau_{\mathsf{aa}}(\mathsf{aa})$, i.e., $\tau_{\mathsf{d}}(\mathsf{aa_t}(\mathsf{aa}))$ is a subtype of $\tau_{\mathsf{aa}}(\mathsf{aa}).\mathsf{eType}$.
3. Reference multiplicity condition: the total number of assignments to a reference $\mathbb{rf}$ in an object should fall in the range specified by the multiplicity of $\mathbb{rf}$. Namely, $\forall \mathsf{o} \in \mathsf{O}, \mathbb{rf} \in \mathbb{Rf}$, let $k = \sharp\{\mathsf{ra} \in \mathsf{RA} \mid \tau_{\mathsf{ra}}(\mathsf{ra}) = \mathbb{rf} \text{ and } \mathsf{ra_s}(\mathsf{ra}) = \mathsf{o}\}$ (i.e., the number of assignments to the reference $\mathbb{rf}$ in object o), we must have $k \geq \mathbb{rf}.\mathsf{lowerBound}$ and $k \leq \mathbb{rf}.\mathsf{upperBound}$ in case $\mathbb{rf}.\mathsf{upperBound} \neq -1$.
4. Attribute multiplicity condition: the total number of assignments to an attribute $\mathbb{at}$ in an object should fall in the range specified by the multiplicity of $\mathbb{at}$. Namely, $\forall \mathsf{o} \in \mathsf{O}, \mathbb{at} \in \mathbb{At}$, let $k = \sharp\{\mathsf{aa} \in \mathsf{AA} \mid \tau_{\mathsf{aa}}(\mathsf{aa}) = \mathbb{at} \text{ and } \mathsf{aa_s}(\mathsf{aa}) = \mathsf{o}\}$ (i.e., the number of assignments to the attribute $\mathbb{at}$ in object o), we must have $k \geq \mathbb{at}.\mathsf{lowerBound}$ and $k \leq \mathbb{at}.\mathsf{upperBound}$ in case $\mathbb{at}.\mathsf{upperBound} \neq -1$.
5. Opposite reference condition: opposite references are assigned in a pair-wise way. Namely, given two references $\mathbb{rf}_1$ and $\mathbb{rf}_2$ where $\mathbb{rf}_1.\mathsf{eOpposite} = \mathbb{rf}_2$, for any reference assignment $\mathsf{ra}_1 \in \mathsf{RA}$ where $\tau_{\mathsf{ra}}(\mathsf{ra}_1) = \mathbb{rf}_1$, there exists a reference assignment $\mathsf{ra}_2 \in \mathsf{RA}$ such that $\tau_{\mathsf{ra}}(\mathsf{ra}_2) = \mathbb{rf}_2$, $\mathsf{ra_s}(\mathsf{ra}_1) = \mathsf{ra_t}(\mathsf{ra}_2)$, and $\mathsf{ra_t}(\mathsf{ra}_1) = \mathsf{ra_s}(\mathsf{ra}_2)$.

6. Containment condition: an object cannot be contained in more than one object, neither can it be contained in itself.
7. Abstract class condition: no objects can be instantiated from an abstract class.
8. Uniqueness condition: if a reference (resp. an attribute) is flagged as unique, the assigned values to the reference (resp. the attribute) in an object must be distinct from one to another.
9. Invariant condition: all the invariants should hold.

5 CoreOCL: A Formal Capture of Core EssentialOCL

"EssentialOCL is the package exposing the minimal OCL required to work with EMOF." (OMG OCL specification [27], p. 187). As a consequence, concepts such as messages, association classes, and states are not part of EssentialOCL.

In this section, we formalize a core of EssentialOCL, called CoreOCL. CoreOCL provides a theoretical framework for defining invariants and discussing their evaluation semantics. CoreOCL is the formal abstraction and mathematical foundation of EssentialOCL. CoreOCL consists of the core constructs of EssentialOCL and exposes the same expressive power as the latter. The relationship between EssentialOCL and CoreOCL can be characterized by the following keywords: abstraction, normalization, simplification, and formalization.

Abstraction Pre-defined operations in the OCL standard library that are relevant for EssentialOCL are abstracted into kappa operations, i.e., we use κ to range over their names in the syntax. Kappa operations include pre-defined operations for primitive types such as arithmetic operations and logical operations; pre-defined operations on collection types such as collection size and collection union; and pre-defined operations for all values such as equality-check operations and operations whose names come with an ocl prefix (e.g., $\mathsf{oclIsUndefined}()$). Note that kappa operations take only values as parameters. In case a pre-defined operation in the OCL standard library takes also types as parameters, such as the $\mathsf{oclAsType}$ and $\mathsf{allInstances}$ operations, they are captured in CoreOCL explicitly.

Normalization A uniform prefix notation is used for all kappa operation calls. For example, $x \wedge y$ is expressed as $\mathsf{AND}(x, y)$, $x > y$ is expressed as $>(x, y)$, and $\mathsf{Set}\{\} \rightarrow \mathsf{including}(x)$ is expressed as $\mathsf{including}(\mathsf{Set}\{\}, x)$, where AND, $>$, and $\mathsf{including}$ are all instances of κ.

Moreover, for each language concept, CoreOCL supports only one canonical syntactic representation. Inessential "syntactic sugar", i.e., syntax constructs that can be removed without any effect on the expressive power, are dismissed in CoreOCL. Applications of syntactic sugar can be systematically replaced with equivalent constructs from the core subset. Here are some examples of dis-

missed syntactic sugar and the corresponding encoding in CoreOCL:

- Tuple: the type information in a tuple is required in legal CoreOCL syntax, and all tuples without types specified for their parts, which is legal syntax in EssentialOCL, should be augmented with the inferred type information. For example, the following tuple: Tuple(name="Smith Johns", age=30) will be expressed as Tuple(name:String="Smith Johns", age: Integer=30).
- Collection iterator: only the iterate operation is legal syntax in CoreOCL (see CoreOCL syntax below), and all other collection iterators need to be described in terms of iterate. For example, the following iterator: forAll(x| $>(x, 0)$), checks if a set has only positive integers. This will be expressed as iterate($x; y =$ true | if $>(x, 1)$ then AND(y, true) else AND(y, false)).
- Non-empty collection literals: only empty collection literals are legal syntax in CoreOCL (see CoreOCL syntax below), and all non-empty collection literals such as Set$\{1, 2\}$ will be expressed as nested calls to the pre-defined "including" operation on collections, i.e., including(including(Set$\{\}, 1), 2$).

Simplification CoreOCL omits the following in the syntax: OclAny, OclVoid, OclInvalid, null, and invalid, because they are not supposed to be used directly in expressions. The first three types are used in the type system of CoreOCL, and the last two values are used in the evaluation semantics of CoreOCL.

Formalization CoreOCL lays a formal foundation upon which we can precisely define the evaluation semantics of invariants written in CoreOCL and the corresponding scope that has been introduced abstractly in Definition 6. Moreover, we are also able to demonstrate (in Theorem 5) formally that letting a sub-model include all the instances of the evaluation scope of an invariant as they are in the original model suffices to maintain the same evaluation, whereas in Section 3, this statement can only be discussed informally given the abstract setting. Finally, we explore a sufficient condition for identifying forward invariants as defined in Definition 8 in a practical way, by proving that CoreOCL invariants that do not call allInstances are all forward.

We present CoreOCL in the context of EMF. Namely, the concrete syntax of CoreOCL adopts EMF notations for the specification of invariants for EMF metamodels and the semantics of CoreOCL is defined in terms of how these invariants are evaluated on EMF models.

5.1 CoreOCL expressions: syntax

The detailed definition of CoreOCL expressions is given below in EBNF format.

$\mathbb{t}$ ::=		**Types**
	$\mathbb{dt}$	Data types
	$\mathbb{c}$	Classes
	Collection($\mathbb{t}$)	Collection types
	Bag($\mathbb{t}$)	Bag types
	Set($\mathbb{t}$)	Set types
	Seq($\mathbb{t}$)	Sequence types
	OSet($\mathbb{t}$)	Ordered set types
	Tuple($\overline{\mathbb{tn} : \mathbb{t}}$)	Tuple types
$\mathbb{e}$ ::=		**Expressions**
	dv	Data value
	x | self	Variable
	$\mathbb{e}.\mathbb{sf}$	Structural feature call
	Tuple{$\overline{\mathbb{tn} : \mathbb{t} = \mathbb{e}}$}	Tuple
	$\mathbb{e}.\mathbb{tn}$	Tuple part call
	let $x = \mathbb{e}_1$ in $\mathbb{e}_2$	Let binding
	def $\mathbb{c} :: \mathbb{op} = \lambda(\overline{x} : \overline{\mathbb{t}}).\mathbb{e}_1$ in $\mathbb{e}_2$	Def expression
	$\mathbb{e}.\mathbb{op}(\overline{\mathbb{e}})$	Object operation call
	$\kappa(\overline{\mathbb{e}})$	Kappa operation call
	if $\mathbb{e}_1$ then $\mathbb{e}_2$ else $\mathbb{e}_3$	Conditional expression
	$\mathbb{e}$ asInstanceOf $\mathbb{c}$	Downcast
	$\mathbb{e}$ isInstanceOf $\mathbb{c}$	Is Instance Of
	$\mathbb{e}$ isOfKind $\mathbb{c}$	Is Kind Of
	Bag{} | Set{} | Seq{} | OSet{}	Empty collections
	$\mathbb{e}_1 \rightarrow$ iterate($x; y = \mathbb{e}_2 \mid \mathbb{e}_3$)	Collection iteration
	allInstances($\mathbb{c}$)	All instances

The following notational convention is adopted for sequences: we write $\overline{X}$ as the shorthand for a sequence of the form $X_1, \ldots, X_k$, when the members of the sequence are not necessarily required to be accessible explicitly. We also abbreviate handling of paired sequences in an obvious way. For example, we write $(\overline{x} : \overline{\mathbb{t}})$ as the shorthand for $(x_1 : \mathbb{t}_1, \ldots, x_k : \mathbb{t}_k)$, $(\overline{\mathbb{tn} : \mathbb{t} = \mathbb{e}})$ as the shorthand for $(\mathbb{tn}_1 : \mathbb{t}_1 = \mathbb{e}_1, \ldots, \mathbb{tn}_k : \mathbb{t}_k = \mathbb{e}_k)$, and $(\overline{x} := \overline{\mathsf{v}})$ as the short hand for $(x_1 := \mathsf{v}_1, \ldots, x_k := \mathsf{v}_k)$. The first is used in the syntax to specify the parameters and their types of an object operation. The second is used in the syntax to specify the parts of a tuple. And the third is used in the semantic rule for evaluating object operation calls to assign arguments to parameters (See rule ObjectOperationCall in Figure 6).

In addition to data types and classes, OCL introduces the notion of collections and tuples. Collections are used to handle navigations of multi-valued structural features (i.e., references, attributes, etc.). A structural feature is *single-valued* when its upper bound is 1, otherwise *multi-valued*. Five collection types are provided, where four of them, i.e., the Bag, Set, Seq, and OSet types are concrete and the fifth, i.e., *Collection* is the abstract supertype of the other four. Moreover, both Set and Seq are subtypes of Bag, and OSet is a subtype of both Set and Seq.

Tuple types (resp. values) provide a way to compose several types (resp. values) together to form compound types (resp. values). The parts of a tuple can be accessed by their names (ranged over by $\mathbb{tn}$) using the same dot notation as is used for accessing structural features.

Variables are ranged over by identifiers such as x, y, etc. The keyword self is a reserved identifier that represents a special variable referring to the current contextual object. We use $\mathbb{sf}$ to range over both attributes and references, which are commonly referred to as structural features in EMF (and properties in EMOF). We can call a structural feature on an object to retrieve the assigned value(s).

Let bindings and def expressions define variables and object operations (ranged over by $\mathbb{op}$) reusable in the nested expressions. Note that according to [27], all the object operations used in OCL are side effect free. The parameters of an operation are treated as variables local to the corresponding body expression of the operation. We call an operation on an object by passing concrete arguments to parameters.

Finally, iterator operations are a special type of operations on collection types that enable one to iterate over the elements in a collection. We only capture the most fundamental and generic collection iterator called iterate in CoreOCL to keep the language small, as all the other collection iterators can be derived from it [34].

5.2 CoreOCL expressions: evaluation

5.2.1 Evaluation judgments

The evaluation of a CoreOCL expression $\mathbb{e}$ takes place in a model EM and an evaluation environment Λ and results in a pair $(\mathsf{v}, \mathsf{O}_S)$, where v is the value of the expression and O_S consists of the objects of EM that are referenced by $\mathbb{e}$ during the evaluation. The model EM is exploited during the course of the evaluation for relevant model information such as the types of instances and the values assigned to attributes and references.

Thereafter, we make use of the following judgment to denote the evaluation semantics of CoreOCL expressions:

$$\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e} \Downarrow (\mathsf{v}, \mathsf{O}_S)$$

which we read as: within the context Λ and the model EM, the value of $\mathbb{e}$ is $(\mathsf{v}, \mathsf{O}_S)$.

5.2.2 Evaluation environments

An evaluation environment Λ is a function that relates qualified object operation names ($\mathbb{c} :: \mathbb{op}$) to operation definitions ($\lambda(\overline{x}).\mathbb{e}$) and variables to values. In CoreOCL, we consider the following types of values:

v ::=	**Values**
dv	Data value
o	Object value
Tuple{$\overline{\mathbb{tn} : \mathbb{t} = \mathsf{v}}$}	Tuple value
null	Unknown value
Bag{$\overline{\mathsf{v}}$} | Set{$\overline{\mathsf{v}}$} | Seq{$\overline{\mathsf{v}}$} | OSet{$\overline{\mathsf{v}}$}	Collection value

Evaluation environments can be specified extensionally by listing the pairs related in it. The overriding of an evaluation environment Λ_1 by another evaluation environment Λ_2 (also called overriding union) is an extension of Λ_2 denoted as $\Lambda_1 \oplus \Lambda_2$. The domain of $\Lambda_1 \oplus \Lambda_2$ is the union of the domain of Λ_1 and the domain of Λ_2. In $\Lambda_1 \oplus \Lambda_2$, any element of the domain of Λ_2 is related to

its image under Λ_2, and any other element exclusive to the domain of Λ_1 is related to its image under Λ_1.

5.2.3 Expression evaluation rules

We say an evaluation judgment holds if a derivation tree of the evaluation judgment can be constructed by instantiating the evaluation rules presented in Figure 6. Note that in the presentation of the rules, wherever the model information is required for a computation, EM appears explicitly on top of the corresponding computation sign. Given an evaluation environment Λ, an expression $\mathbb{e}$ and a model EM, if no evaluation judgment holds for them, we say $\mathbb{e}$ is *invalid* within context Λ and model EM.

Some words about typing Without making the type system explicit, the evaluation rules assign only semantics to well-typed expressions. Notably, we draw the attention of the reader to the following points.

1. Due to subtype polymorphism, an expression of type $\mathbb{t}$ has also all the super types of $\mathbb{t}$.
2. Variables are first defined via let bindings then referenced in nested variable expressions. Note that being a pre-defined variable for the reference of the current contextual object, no explicit let binding is needed for self.
3. Operations defined via def expressions carry distinct parameters.
4. Structural feature calls and object operation calls are only addressed on objects and the callee (i.e., a reference or an attribute or an object operation) is actually defined for the type of the object.

   Subsequently, calling structural features or object operations on a collection of objects should be explicitly implemented by using the iterate operation that iterates the structural feature or object operation call on all the element objects of the collection.
5. Tuple part calls are only addressed on tuple values and the callee (i.e., the called tuple part name $\mathbb{tn}$) is actually a defined part of the tuple.

   Similarly, calling parts on a collection of tuples should be explicitly implemented by using the iterate operation.
6. Given a tuple $\mathsf{Tuple}\{\overline{\mathbb{tn}} : \overline{\mathbb{t}} = \overline{\mathbb{e}}\}$, the type of the expression $\mathbb{e}_i$ that is used to indicate the value of the part $\mathbb{tn}_i$ in the tuple is compatible with the type that is specified for the part in the tuple, i.e., $\mathbb{t}_i$.
7. Both object operations and kappa operations are called with type and cardinality compatible arguments.
8. Given a conditional expression $\mathsf{if}\ \mathbb{e}_1\ \mathsf{then}\ \mathbb{e}_2\ \mathsf{else}\ \mathbb{e}_3$, $\mathbb{e}_1$ has type Boolean, and $\mathbb{e}_2$ and $\mathbb{e}_3$ have the same type.
9. Downcast expressions (AsInstanceOf), type testing expressions (isInstanceOf), and subtype testing expressions (isKindOf) are only applied to objects.
10. Finally, given a collection iteration expression $\mathbb{e}_1 \rightarrow \mathsf{iterate}(x; y = \mathbb{e}_2 \mid \mathbb{e}_3)$, $\mathbb{e}_1$ has a collection type. More-

DataValue

$$\Lambda \overset{\mathsf{EM}}{\vDash} \mathsf{dv} \Downarrow (\mathsf{dv}, \emptyset)$$

Variable

$$\frac{\Lambda(x) = \mathsf{v} \qquad \mathsf{O}_S(\mathsf{v}) \subseteq \mathsf{O}}{\Lambda \overset{\mathsf{EM}}{\vDash} x \Downarrow (\mathsf{v}, \mathsf{O}_S(\mathsf{v}))}$$

StructuralFeatureCall

$$\frac{\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e} \Downarrow (\mathsf{o}, \mathsf{O}_S^1) \qquad \mathsf{o}.\mathbb{sf} \overset{\mathsf{EM}}{=} \mathsf{v}}{\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}.\mathbb{sf} \Downarrow (\mathsf{v}, \mathsf{O}_S^1 \cup \mathsf{O}_S(\mathsf{v}))}$$

Tuple

$$\frac{\Lambda \overset{\mathsf{EM}}{\vDash} \overline{\mathbb{e}} \Downarrow (\overline{\mathsf{v}}, \overline{\mathsf{O}_S})}{\Lambda \overset{\mathsf{EM}}{\vDash} \mathsf{Tuple}\{\overline{\mathbb{tn} : \mathbb{t} = \mathbb{e}}\} \Downarrow (\mathsf{Tuple}\{\overline{\mathbb{tn} : \mathbb{t} = \mathsf{v}}\}, \bigcup \overline{\mathsf{O}_S})}$$

TuplePartCall

$$\frac{\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e} \Downarrow (\mathsf{Tuple}\{\overline{\mathbb{tn} : \mathbb{t} = \mathsf{v}}\}, \mathsf{O}_S) \qquad \mathbb{tn}_i = \mathbb{tn}}{\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}.\mathbb{tn} \Downarrow (\mathsf{v}_i, \mathsf{O}_S)}$$

LetBinding

$$\frac{\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}_1 \Downarrow (\mathsf{v}_1, \mathsf{O}_S^1) \qquad \Lambda \oplus \{(x : \mathsf{v}_1)\} \overset{\mathsf{EM}}{\vDash} \mathbb{e}_2 \Downarrow (\mathsf{v}_2, \mathsf{O}_S^2)}{\Lambda \overset{\mathsf{EM}}{\vDash} \mathsf{let}\ x = \mathbb{e}_1\ \mathsf{in}\ \mathbb{e}_2 \Downarrow (\mathsf{v}_2, \mathsf{O}_S^1 \cup \mathsf{O}_S^2)}$$

DefExp

$$\frac{\Lambda \oplus \{(\mathbb{op} : \lambda(\overline{x}).\mathbb{e}_1)\} \overset{\mathsf{EM}}{\vDash} \mathbb{e}_2 \Downarrow (\mathsf{v}, \mathsf{O}_S)}{\Lambda \overset{\mathsf{EM}}{\vDash} \mathsf{def}\ \mathbb{c} :: \mathbb{op} = \lambda(\overline{x} : \overline{\mathbb{t}}).\mathbb{e}_1\ \mathsf{in}\ \mathbb{e}_2 \Downarrow (\mathsf{v}, \mathsf{O}_S)}$$

ObjectOperationCall

$$\frac{\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e} \Downarrow (\mathsf{o}, \mathsf{O}_S^{\mathsf{o}}) \qquad \Lambda \overset{\mathsf{EM}}{\vDash} \overline{\mathbb{e}} \Downarrow (\overline{\mathsf{v}}, \overline{\mathsf{O}_S}) \qquad \Lambda(\mathbb{op}) = \lambda(\overline{x}).\mathbb{e}_{\mathbb{op}} \qquad \Lambda \oplus \{(\mathsf{self} : \mathsf{o})\} \oplus \{(\overline{x} : \overline{\mathsf{v}})\} \overset{\mathsf{EM}}{\vDash} \mathbb{e}_{\mathbb{op}} \Downarrow (\mathsf{v}, \mathsf{O}_S^{\mathbb{op}})}{\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}.\mathbb{op}(\overline{x} := \overline{\mathbb{e}}) \Downarrow (\mathsf{v}, \mathsf{O}_S^{\mathsf{o}} \cup (\bigcup \overline{\mathsf{O}_S}) \cup \mathsf{O}_S^{\mathbb{op}})}$$

KappaCall

$$\frac{\Lambda \overset{\mathsf{EM}}{\vDash} \overline{\mathbb{e}} \Downarrow (\overline{\mathsf{v}}, \overline{\mathsf{O}_S}) \qquad \kappa(\overline{\mathsf{v}}) = \mathsf{v}}{\Lambda \overset{\mathsf{EM}}{\vDash} \kappa(\overline{\mathbb{e}}) \Downarrow (\mathsf{v}, \bigcup \overline{\mathsf{O}_S})}$$

IfTrue

$$\frac{\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}_1 \Downarrow (\mathsf{true}, \mathsf{O}_S^1) \qquad \Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}_2 \Downarrow (\mathsf{v}, \mathsf{O}_S^2)}{\Lambda \overset{\mathsf{EM}}{\vDash} \mathsf{if}\ \mathbb{e}_1\ \mathsf{then}\ \mathbb{e}_2\ \mathsf{else}\ \mathbb{e}_3 \Downarrow (\mathsf{v}, \mathsf{O}_S^1 \cup \mathsf{O}_S^2)}$$

IfFalse

$$\frac{\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}_1 \Downarrow (\mathsf{false}, \mathsf{O}_S^1) \qquad \Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}_3 \Downarrow (\mathsf{v}, \mathsf{O}_S^2)}{\Lambda \overset{\mathsf{EM}}{\vDash} \mathsf{if}\ \mathbb{e}_1\ \mathsf{then}\ \mathbb{e}_2\ \mathsf{else}\ \mathbb{e}_3 \Downarrow (\mathsf{v}, \mathsf{O}_S^1 \cup \mathsf{O}_S^2)}$$

DowncastOK

$$\frac{\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e} \Downarrow (\mathsf{o}, \mathsf{O}_S) \qquad \tau_{\mathsf{o}}(\mathsf{o}) \preccurlyeq \mathbb{c}}{\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}\ \mathsf{asInstanceOf}\ \mathbb{c} \Downarrow (\mathsf{o}, \mathsf{O}_S)}$$

DowncastNotOK

$$\frac{\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e} \Downarrow (\mathsf{o}, \mathsf{O}_S) \qquad \tau_{\mathsf{o}}(\mathsf{o}) \not\preccurlyeq \mathbb{c}}{\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}\ \mathsf{asInstanceOf}\ \mathbb{c} \Downarrow (\mathsf{null}, \mathsf{O}_S)}$$

IsInstanceOfTrue

$$\frac{\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e} \Downarrow (\mathsf{o}, \mathsf{O}_S) \qquad \tau_{\mathsf{o}}(\mathsf{o}) \overset{\mathsf{EM}}{=} \mathbb{c}}{\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}\ \mathsf{isInstanceOf}\ \mathbb{c} \Downarrow (\mathsf{true}, \mathsf{O}_S)}$$

IsInstanceOfFalse

$$\frac{\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e} \Downarrow (\mathsf{o}, \mathsf{O}_S) \qquad \tau_{\mathsf{o}}(\mathsf{o}) \overset{\mathsf{EM}}{\neq} \mathbb{c}}{\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}\ \mathsf{isInstanceOf}\ \mathbb{c} \Downarrow (\mathsf{false}, \mathsf{O}_S)}$$

IsKindOfTrue

$$\frac{\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e} \Downarrow (\mathsf{o}, \mathsf{O}_S) \qquad \tau_{\mathsf{o}}(\mathsf{o}) \overset{\mathsf{EM}}{=} \mathbb{c}' \qquad \mathbb{c}' \preccurlyeq \mathbb{c}}{\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}\ \mathsf{isOfKind}\ \mathbb{c} \Downarrow (\mathsf{true}, \mathsf{O}_S)}$$

IsKindOfFalse

$$\frac{\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e} \Downarrow (\mathsf{o}, \mathsf{O}_S) \qquad \tau_{\mathsf{o}}(\mathsf{o}) \overset{\mathsf{EM}}{=} \mathbb{c}' \qquad \mathbb{c}' \not\preccurlyeq \mathbb{c}}{\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}\ \mathsf{isOfKind}\ \mathbb{c} \Downarrow (\mathsf{false}, \mathsf{O}_S)}$$

EmptyBag

$$\Lambda \overset{\mathsf{EM}}{\vDash} \mathsf{Bag}\{\} \Downarrow (\mathsf{Bag}\{\}, \emptyset)$$

EmptySet

$$\Lambda \overset{\mathsf{EM}}{\vDash} \mathsf{Set}\{\} \Downarrow (\mathsf{Set}\{\}, \emptyset)$$

CollectionIteration

$$\frac{\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}_1 \Downarrow (\mathit{Collection}\{\mathsf{v}'_1, \ldots, \mathsf{v}'_k\}, \mathsf{O}_S^0) \qquad \Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}_2 \Downarrow (\mathsf{v}_1, \mathsf{O}_S^1) \qquad (\Lambda \oplus \{(x : \mathsf{v}'_i)\} \oplus \{(y : \mathsf{v}_i)\} \overset{\mathsf{EM}}{\vDash} \mathbb{e}_3 \Downarrow (\mathsf{v}_{i+1}, \mathsf{O}_S^{i+1}))^{\,i=1,\ldots,k}}{\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}_1 \to \mathsf{iterate}(x; y = \mathbb{e}_2 \mid \mathbb{e}_3) \Downarrow (\mathsf{v}_{k+1}, \bigcup_{0 \le i \le k+1} \mathsf{O}_S^i)}$$

EmptySeq

$$\Lambda \overset{\mathsf{EM}}{\vDash} \mathsf{Seq}\{\} \Downarrow (\mathsf{Seq}\{\}, \emptyset)$$

EmptyOSet

$$\Lambda \overset{\mathsf{EM}}{\vDash} \mathsf{OSet}\{\} \Downarrow (\mathsf{OSet}\{\}, \emptyset)$$

AllInstances

$$\Lambda \overset{\mathsf{EM}}{\vDash} \mathsf{allInstances}(\mathbb{c}) \Downarrow (\mathsf{Set}\{\mathsf{o} \mid \mathsf{o} \in \mathsf{O} \text{ and } \tau_{\mathsf{o}}(\mathsf{o}) \overset{\mathsf{EM}}{=} \mathbb{c}\}, \{\mathsf{o} \mid \mathsf{o} \in \mathsf{O} \text{ and } \tau_{\mathsf{o}}(\mathsf{o}) \overset{\mathsf{EM}}{=} \mathbb{c}\})$$

Fig. 6 Evaluation rules for CoreOCL expressions

over, the expression $\mathbb{e}_2$ that assigns initial value to the accumulator y and the body expression $\mathbb{e}_3$ have the same type, in order for the iteration to take place properly.

Digesting the rules Rules StructuralFeatureCall and Variable in Figure 6 both rely on an auxiliary definition $\mathsf{O}_S(\mathsf{v})$ that computes the set of objects directly referenced by a value v. The computation is defined recursively for different kinds of values as follows:

$$\mathsf{O}_S(\mathsf{dv}) = \emptyset$$
$$\mathsf{O}_S(\mathsf{o}) = \{\mathsf{o}\}$$
$$\mathsf{O}_S(\mathsf{Tuple}\{\overline{\mathbb{tn}} : \overline{\mathbb{t}} = \overline{\mathsf{v}}\}) = \bigcup_{\mathsf{v}_i \in \overline{\mathsf{v}}} \mathsf{O}_S(\mathsf{v}_i)$$
$$\mathsf{O}_S(\mathsf{null}) = \emptyset$$
$$\mathsf{O}_S(\mathsf{Bag}\{\overline{\mathsf{v}}\}) = \bigcup_{\mathsf{v}_i \in \overline{\mathsf{v}}} \mathsf{O}_S(\mathsf{v}_i)$$
$$\mathsf{O}_S(\mathsf{Set}\{\overline{\mathsf{v}}\}) = \bigcup_{\mathsf{v}_i \in \overline{\mathsf{v}}} \mathsf{O}_S(\mathsf{v}_i)$$
$$\mathsf{O}_S(\mathsf{Seq}\{\overline{\mathsf{v}}\}) = \bigcup_{\mathsf{v}_i \in \overline{\mathsf{v}}} \mathsf{O}_S(\mathsf{v}_i)$$
$$\mathsf{O}_S(\mathsf{OSet}\{\overline{\mathsf{v}}\}) = \bigcup_{\mathsf{v}_i \in \overline{\mathsf{v}}} \mathsf{O}_S(\mathsf{v}_i)$$

The second premise of rule Variable requires that the value v associated to the variable x in the evaluation environment should only reference objects of EM i.e., O, in order for the value to be meaningful.

The computation of a structural feature on an object in a model is present as the second premises of rule StructuralFeatureCall. A structural feature is either a reference or an attribute. We write $\mathsf{n}.\mathbb{sf} \overset{\mathsf{EM}}{=} \mathsf{v}$ to denote the computation of a well-typed structural feature call in a model, whose result is either a collection, a data value, an object, or null, depending on the target multiplicity of the structural feature. More specifically, if the structural feature is single-valued, i.e., its upper bound is 1, the result is the data value assigned to the attribute or the object assigned to the reference, or null if no value is assigned to the structural feature at all. Otherwise, the result is a collection of the values assigned to the structural feature in the model. Then depending on the two flags: ordered and unique, the collection is either a set (not ordered but unique), a bag (not ordered and not unique), a sequence (ordered but not unique), or a ordered set (ordered and unique).

We summarize the computation of a structural feature call on an object in a model in Table 1, in which, we use sfa to range over $\mathsf{RA} \cup \mathsf{AA}$, i.e., assignments to structural features in a model EM, and use $\mathsf{sfa}.t$ to denote the target of the assignment, i.e., if sfa is a reference assignment then $\mathsf{ra_t}(\mathsf{sfa})$ and if sfa is an attribute assignment then $\mathsf{aa_t}(\mathsf{sfa})$.

In rule ObjectOperationCall, calling an operation $\mathbb{op}$ on an object o evaluates the body expression of the operation $\mathbb{e}_{\mathbb{op}}$ with the parameters $\overline{x}$ replaced by proper arguments.

The evaluation of $\kappa(\overline{\mathsf{v}})$ in rule KappaCall is defined by the semantics of the pre-defined operation κ. For example, we have $\mathsf{isDefined}(\mathsf{null}) = \mathsf{false}$, $+(1, 2) = 3$, and $\mathsf{size}(\mathsf{Set}\{\}) = 0$, etc.

Given a call to the iterate operation $\mathbb{e}_1 \rightarrow \mathsf{iterate}(x; y = \mathbb{e}_2 \mid \mathbb{e}_3)$, expression $\mathbb{e}_1$ should have a collection value of either the four collection types: Bag, Set, Seq, and OSet. In rule CollectionIteration, we write the abstract value

<table>
<tr><th>o.𝕤𝕗 $\overset{\mathsf{EM}}{=}$?</th><th colspan="4">𝕤𝕗 multi-valued
(i.e.,𝕤𝕗.upperBound > 1 or $= -1$)</th><th>𝕤𝕗 single-valued
(i.e., 𝕤𝕗.upperBound $= 1$)</th></tr>
<tr><td rowspan="3">∃ assignments to 𝕤𝕗 for o in EM</td><td colspan="2">𝕤𝕗.ordered = true
Let ($\mathsf{sfa}_1, \ldots, \mathsf{sfa}_n$) be the chain of assignments to 𝕤𝕗 for o in EM (see Section 4.2)</td><td colspan="2">𝕤𝕗.ordered = false
Let $\{\mathsf{sfa}_1, \ldots, \mathsf{sfa}_n\}$ be the set of assignments to 𝕤𝕗 for o in EM (see Section 4.2)</td><td rowspan="3">The value assigned to 𝕤𝕗 for o in EM</td></tr>
<tr><td>𝕤𝕗.unique = true</td><td>𝕤𝕗.unique = false</td><td>𝕤𝕗.unique = true</td><td>𝕤𝕗.unique = false</td></tr>
<tr><td>OSet{$\mathsf{sfa}_1.t, \ldots, \mathsf{sfa}_n.t$}</td><td>Seq{$\mathsf{sfa}_1.t, \ldots, \mathsf{sfa}_n.t$}</td><td>Set{$\mathsf{sfa}_1.t, \ldots, \mathsf{sfa}_n.t$}</td><td>Bag{$\mathsf{sfa}_1.t, \ldots, \mathsf{sfa}_n.t$}</td></tr>
<tr><td>otherwise</td><td>OSet{}</td><td>Seq{}</td><td>Set{}</td><td>Bag{}</td><td>null</td></tr>
</table>

Table 1 Calling structural features (i.e., attributes or references) on an object in a model

Collection{$\overline{\mathsf{v}}$} to stand for any of the four kinds of collection values. The result of calling the iterate operation on *Collection*{$\overline{\mathsf{v}}$} is a value obtained by iterating over all elements in the collection. The variable x is the *iterator*. It goes through the elements in the source collection for each round of the iteration. The variable y is the *accumulator*. It gets an initial value given by expression $\mathbb{e}_2$ and is used to accumulate results during the iteration. For each element in the source collection referenced by the iterator x, the body expression $\mathbb{e}_3$ is calculated using the current value of the accumulator y, and the result is assigned back to the accumulator for the next iteration until the last element of the collection. A simple example of collection iteration is given below to calculate the sum of the elements of a set of integers:

$$\mathsf{Set}\{1,2,3\} \rightarrow \mathsf{iterate}(x; y = 0 \,|\, x + y)$$

5.3 CoreOCL invariants

An invariant specified in CoreOCL, written $\mathbb{e}\mathbb{i}\mathbb{n}\mathbb{v} = (\mathbb{c}, \mathbb{e})$, comprises a context class $\mathbb{c}$ and a boolean CoreOCL expression $\mathbb{e}$ being the body of the invariant.

To check if a model EM satisfies all the invariants specified for its metamodel, we need to evaluate all the well-formed evaluation points of the form $(\mathbb{e}\mathbb{i}\mathbb{n}\mathbb{v}, \mathsf{EM}, \mathsf{o})$ (see Definition 5), where $\mathbb{e}\mathbb{i}\mathbb{n}\mathbb{v}$ is an invariant defined for the metamodel of EM, o is an object of EM, and the type of o is a subtype of the context of $\mathbb{e}\mathbb{i}\mathbb{n}\mathbb{v}$. Evaluating such a well-formed evaluation point amounts to evaluating the boolean body expression of $\mathbb{e}\mathbb{i}\mathbb{n}\mathbb{v}$ within the model EM and *an initial evaluation environment*. The domain of the initial evaluation environment consists of the keyword self associated with the current contextual object o. We write $\mathsf{eval}(\mathbb{e}\mathbb{i}\mathbb{n}\mathbb{v}, \mathsf{EM}, \mathsf{o})$ to denote the result of evaluating $(\mathbb{e}\mathbb{i}\mathbb{n}\mathbb{v}, \mathsf{EM}, \mathsf{o})$.

The first part of the result, which is a boolean value, tells whether the invariant $\mathbb{inv}$ holds for the contextual object o in model EM, and the second part of the result is used for the construction of the scope of $(\mathbb{inv}, \mathsf{EM}, \mathsf{o})$, which is an object induced sub-model of EM following the definition below.

Definition 10 (Object induced EMF sub-model) *We say an EMF model* $\mathsf{EM}' = (\mathbb{EM}, \mathsf{O}', \tau_{\mathsf{o}}', \mathsf{RA}', \tau_{\mathsf{ra}}', \mathsf{ra_t}', \mathsf{ra_s}', \mathsf{AA}', \tau_{\mathsf{aa}}', \mathsf{aa_s}', \mathsf{aa_t}')$ *is an object induced sub-model (shortly sub-model) of an EMF model* $\mathsf{EM} = (\mathbb{EM}, \mathsf{O}, \tau_{\mathsf{o}}, \mathsf{RA}, \tau_{\mathsf{ra}}, \mathsf{ra_t}, \mathsf{ra_s}, \mathsf{AA}, \tau_{\mathsf{aa}}, \mathsf{aa_s}, \mathsf{aa_t})$*, if and only if:*

1. $\mathsf{O}' \subseteq \mathsf{O}$;
2. τ_{o}' *is the restriction of* τ_{o} *to* O';
3. $\mathsf{RA}' = \{\mathsf{ra} \mid \mathsf{ra} \in \mathsf{RA}, \mathsf{ra_s}(\mathsf{ra}) \in \mathsf{O}', \mathsf{ra_t}(\mathsf{ra}) \in \mathsf{O}'\}$;
4. τ_{ra}', $\mathsf{ra_s}'$ *and* $\mathsf{ra_t}'$ *are respectively the restriction of* τ_{ra}, $\mathsf{ra_s}$, *and* $\mathsf{ra_t}$ *to* RA';
5. *The partial order over* RA *in* EM *is restricted to* RA';
6. $\mathsf{AA}' = \{\mathsf{aa} \mid \mathsf{aa} \in \mathsf{AA}, \mathsf{aa_s}(\mathsf{aa}) \in \mathsf{O}'\}$;
7. τ_{aa}', $\mathsf{aa_s}'$ *and* $\mathsf{aa_t}'$ *are respectively the restriction of* τ_{aa}, $\mathsf{aa_s}$, *and* $\mathsf{aa_t}$ *to* AA';
8. *The partial order over* AA *in* EM *is restricted to* AA'.

5.4 Scopes determine invariant evaluations

We first prove in the following lemma that the evaluation of a CoreOCL expression only involves the objects of the context model in which the expression is being evaluated.

Lemma 3 *If an evaluation judgment* $\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e} \Downarrow (\mathsf{v}, \mathsf{O}_S)$ *holds, then* $\mathsf{O}_S(\mathsf{v}) \subseteq \mathsf{O}_S$.

Proof Straightforward by induction on the structure of expression $\mathbb{e}$. □

Lemma 4 *Given a model* EM*, an evaluation environment* Λ*, and an expression of CoreOCL* $\mathbb{e}$*, if* $\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e} \Downarrow (\mathsf{v}, \mathsf{O}_S)$ *holds, then for any object induced sub-model* EM' *of* EM *where* $\mathsf{O}_S \subseteq \mathsf{O}'$*,* $\Lambda \overset{\mathsf{EM}'}{\vDash} \mathbb{e} \Downarrow (\mathsf{v}, \mathsf{O}_S)$ *also holds.*

Proof See Appendix A.1. □

Theorem 5 *Given two well formed evaluation points:* $(\mathbb{inv}, \mathsf{EM}, \mathsf{o})$ *and* $(\mathbb{inv}, \mathsf{EM}', \mathsf{o})$*, where* EM' *an object induced sub-model of* EM *and* EM' *includes all the objects of the scope of* $(\mathbb{inv}, \mathsf{EM}, \mathsf{o})$*,* $\mathsf{eval}(\mathbb{inv}, \mathsf{EM}, \mathsf{o}) = \mathsf{eval}(\mathbb{inv}, \mathsf{EM}', \mathsf{o})$.

Proof Follows as a corollary of Lemma 4. □

5.5 CoreOCL invariants without allInstances *are forward*

Theorem 6 *Without calling* allInstances*, CoreOCL invariants are all forward following Definition 8.*

Proof See Appendix A.1. □

Lemma 7 *Given a model* EM*, an object* o *in it, a CoreOCL expression* $\mathbb{e}$ *with no calls to* allInstances *(i.e.,* $\mathbb{e}$ *is an expression written in the sub-language of CoreOCL excluding* allInstances*), and an evaluation environment*

Λ in which values bound to variables only refer to objects reachable from o if any, i.e., $\forall x$ in the domain of Λ, $\forall \mathsf{o}' \in \mathsf{O}_S(\Lambda(x))$, o' is reachable from o (see Definition 7). Suppose $\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e} \Downarrow (\mathsf{v}, \mathsf{O}_S)$. We have the following two statements hold:

1. *$\forall \mathsf{o}' \in \mathsf{O}_S$, o' is reachable from o;*
2. *$\forall \mathsf{o}' \in \mathsf{O}_S(\mathsf{v})$, o' is reachable from o.*

Proof See Appendix A.1. □

6 Model Decomposition in EMF

6.1 Abstract EMF for Model Decomposition

In order to exploit the model decomposition technique defined in Section 3 for EMF models, EMF metamodels and EMF models are first abstracted into Definition 1 metamodels and Definition 2 models. Briefly speaking, at the metamodeling level, both classes and data types are mapped to metaclasses in the abstract setting; both references and attributes are mapped to associations; operations are not considered; and invariants are discarded, because invariants are not used in the model decomposition algorithm and hence do not need a counterpart in the abstract setting following the mapping. After the decomposition, we examine whether the invariants keep holding in the sub-models on the EMF side (see Theorem 10 below). At the modeling level, both objects and data values are mapped to instances; and both reference assignments and attribute assignments are mapped to links.

EMF metamodel abstraction Given an EMF metamodel $\mathbb{EM} = (\mathbb{C}, \mathbb{DT}, \mathsf{DV}, \mathbb{Rf}, \mathbb{At})$, mapping $\mathbb{F} = (\mathbb{F}_{\mathbb{N}}, \mathbb{F}_{\mathbb{A}})$ abstracts it into a metamodel, written as $\mathbb{F}(\mathbb{EM}) = \mathbb{M} = (\mathbb{N}, \mathbb{A}, \mathbb{H}, \emptyset, \mathsf{s}, \mathsf{t}, \mu_{\mathsf{s}}, \mu_{\mathsf{t}}, \emptyset)$, if the following properties hold.

1. $\mathbb{C} \cup \mathbb{DT}$ is bijectively mapped to $\mathbb{N}$ by $\mathbb{F}_{\mathbb{N}}$.
2. Subtyping is preserved, i.e., given two classes $\mathbb{c}_1$ and $\mathbb{c}_2$ in $\mathbb{C}$, $\mathbb{F}_{\mathbb{N}}(\mathbb{c}_1) \preccurlyeq \mathbb{F}_{\mathbb{N}}(\mathbb{c}_2)$ if and only if $\mathbb{c}_1 \preccurlyeq \mathbb{c}_2$; given two data types $\mathbb{dt}_1$ and $\mathbb{dt}_2$ in $\mathbb{DT}$, $\mathbb{F}_{\mathbb{N}}(\mathbb{dt}_1) \preccurlyeq \mathbb{F}_{\mathbb{N}}(\mathbb{dt}_2)$ if and only if $\mathbb{dt}_1 \preccurlyeq \mathbb{dt}_2$.
3. $\mathbb{Rf} \cup \mathbb{At}$ is bijectively mapped to $\mathbb{A}$ by $\mathbb{F}_{\mathbb{A}}$.
4. Target types and source types are preserved.

   Given a reference $\mathbb{rf} \in \mathbb{Rf}$,

   (a) $\mathsf{t}(\mathbb{F}_{\mathbb{A}}(\mathbb{rf})) = \mathbb{F}_{\mathbb{N}}(\mathbb{rf}.\mathsf{eType})$;

   (b) $\mathsf{s}(\mathbb{F}_{\mathbb{A}}(\mathbb{rf})) = \mathbb{F}_{\mathbb{N}}(\mathbb{rf}.\mathsf{eContainingClass})$.

   Given an attribute $\mathbb{at} \in \mathbb{At}$,

   (a) $\mathsf{t}(\mathbb{F}_{\mathbb{A}}(\mathbb{at})) = \mathbb{F}_{\mathbb{N}}(\mathbb{at}.\mathsf{eType})$;

   (b) $\mathsf{s}(\mathbb{F}_{\mathbb{A}}(\mathbb{at})) = \mathbb{F}_{\mathbb{N}}(\mathbb{at}.\mathsf{eContainingClass})$.
5. Multiplicities are preserved.

   Given a reference $\mathbb{rf} \in \mathbb{Rf}$,

   (a) the target multiplicity of $\mathbb{F}_{\mathbb{A}}(\mathbb{rf})$ is the same as the multiplicity of $\mathbb{rf}$, i.e., $\mu_{\mathsf{t}}(\mathbb{F}_{\mathbb{A}}(\mathbb{rf})) = (\mathbb{rf}.\mathsf{lowerBound}, \mathbb{rf}.\mathsf{upperBound})$ in case where $\mathbb{rf}.\mathsf{upperBound} \neq -1$, otherwise $\mu_{\mathsf{t}}(\mathbb{F}_{\mathbb{A}}(\mathbb{rf})) = (\mathbb{rf}.\mathsf{lowerBound}, *)$;

   (b) the source multiplicity of $\mathbb{F}_{\mathbb{A}}(\mathbb{rf})$ (i.e. $\mu_{\mathsf{s}}(\mathbb{F}_{\mathbb{A}}(\mathbb{rf}))$) is $(0, 1)$ if $\mathbb{rf}$ is a containment, otherwise $(0, *)$.

   Given an attribute $\mathbb{at} \in \mathbb{At}$,

   (a) the target multiplicity of $\mathbb{F}_{\mathbb{A}}(\mathbb{at})$ is the same as the multiplicity of $\mathbb{at}$, i.e., $\mu_{\mathsf{t}}(\mathbb{F}_{\mathbb{A}}(\mathbb{at})) = (\mathbb{at}.\mathsf{lowerBound},$

$\mathbb{ot}$.upperBound) in case where $\mathbb{ot}$.upperBound $\neq -1$, otherwise $\mu_{\mathsf{t}}(\mathbb{F}_{\mathbb{A}}(\mathbb{ot})) = (\mathbb{ot}.\mathsf{lowerBound}, *)$;

(b) the source multiplicity of $\mathbb{F}_{\mathbb{A}}(\mathbb{ot})$ (i.e. $\mu_{\mathsf{s}}(\mathbb{F}_{\mathbb{A}}(\mathbb{ot}))$) is $(0, *)$.

6. The set of invariants and consequently the invariant context function are empty. Although invariants play a role in determining model conformance (Section 4.3), they are not referenced by the model decomposition algorithm (Section 6.2) hence can be abstracted away.

EMF models abstraction Given an EMF model $\mathsf{EM} = (\mathbb{EM}, \mathsf{O}, \tau_{\mathsf{o}}, \mathsf{RA}, \tau_{\mathsf{ra}}, \mathsf{ra_t}, \mathsf{ra_s}, \mathsf{AA}, \tau_{\mathsf{aa}}, \mathsf{aa_s}, \mathsf{aa_t})$, mapping $\mathsf{F} = (\mathsf{F_N}, \mathsf{F_A})$ abstracts it into a model, written as $\mathsf{F}(\mathsf{EM}) = \mathsf{M} = (\mathbb{M}, \mathsf{N}, \mathsf{A}, \tau, \mathsf{src}, \mathsf{tgt})$, if the following properties hold.

1. $\mathbb{M} = \mathbb{F}(\mathbb{EM})$ is the abstract counterpart of $\mathbb{EM}$.
2. The union of O (from EM) and DV (from $\mathbb{EM}$) is bijectively mapped to N by $\mathsf{F_N}$.
3. $\mathsf{RA} \cup \mathsf{AA}$ is bijectively mapped to A by $\mathsf{F_A}$
4. Typing is preserved, i.e.,
   (a) given an object $\mathsf{o} \in \mathsf{O}$, $\tau(\mathsf{F_N}(\mathsf{o})) = \mathbb{F}_{\mathbb{N}}(\tau_{\mathsf{o}}(\mathsf{o}))$;
   (b) given a data value $\mathsf{dv} \in \mathsf{DV}$, $\tau(\mathsf{F_N}(\mathsf{dv})) = \mathbb{F}_{\mathbb{N}}(\tau_{\mathsf{d}}(\mathsf{dv}))$;
   (c) given a reference assignment $\mathsf{ra} \in \mathsf{RA}$, $\tau(\mathsf{F_A}(\mathsf{ra})) = \mathbb{F}_{\mathbb{A}}(\tau_{\mathsf{ra}}(\mathsf{ra}))$;
   (d) given an attribute assignment $\mathsf{aa} \in \mathsf{AA}$, $\tau(\mathsf{F_A}(\mathsf{aa})) = \mathbb{F}_{\mathbb{A}}(\tau_{\mathsf{aa}}(\mathsf{aa}))$.
5. Source ends and target ends are preserved.
   Given a reference assignment $\mathsf{ra} \in \mathsf{RA}$,
   (a) $\mathsf{src}(\mathsf{F_A}(\mathsf{ra})) = \mathsf{F_N}(\mathsf{ra_s}(\mathsf{ra}))$;
   (b) $\mathsf{tgt}(\mathsf{F_A}(\mathsf{ra})) = \mathsf{F_N}(\mathsf{ra_t}(\mathsf{ra}))$.
   Given an attribute assignment $\mathsf{aa} \in \mathsf{AA}$,
   (a) $\mathsf{src}(\mathsf{F_A}(\mathsf{aa})) = \mathsf{F_N}(\mathsf{aa_s}(\mathsf{aa}))$;
   (b) $\mathsf{tgt}(\mathsf{F_A}(\mathsf{aa})) = \mathsf{F_N}(\mathsf{aa_t}(\mathsf{aa}))$.

Note that in EMF, neither references nor attributes come with a definition for their source multiplicities. The way we give source multiplicities to their images in $\mathbb{M}$ following $\mathbb{F}_{\mathbb{A}}$ (i.e., $(0, *)$ by default and $(0, 1)$ in case of containment references) reflects the least constrained interpretation of the containment relation following the MOF specification [24] (p. 38). As a consequence, all reference assignments and all attribute assignments are fragmentable (in the sense of Definition 9) from the point of view of the assigned values. This allows a fine-degree decomposition wherever possible. However, if two objects are connected by a pair of opposite links, they become indecomposible (see Condition 4 below).

6.2 EMF model decomposition algorithm

Given an EMF model $\mathsf{EM} = (\mathbb{EM}, \mathsf{O}, \tau_{\mathsf{o}}, \mathsf{RA}, \tau_{\mathsf{ra}}, \mathsf{ra_t}, \mathsf{ra_s}, \mathsf{AA}, \tau_{\mathsf{aa}}, \mathsf{aa_s}, \mathsf{aa_t})$, the following algorithm decomposes EM into (object induced) sub-models as defined in Definition 10.

1. Let $\mathsf{M} = \mathsf{F}(\mathsf{EM})$.
2. Apply the model decomposition algorithm as defined in Section 3.2 to M and we get the decomposition hierarchy of M.

3. For any sub-model of M, written M'= ($\mathbb{M}$, N', A', τ', src', tgt'), which is induced by the instances grouped in an antichain-node of the decomposition hierarchy of M, return the corresponding EM', where EM' is a sub-model of EM induced by all the objects of EM whose images following $\mathsf{F_N}$ are included in M, namely $\{\mathsf{o} \mid \mathsf{o} \in \mathsf{O}, \mathsf{F}(\mathsf{o}) \in \mathsf{N}'\}$.

6.3 Soundness of EMF model decomposition

Given a model EM conforming to a metamodel $\mathbb{EM}$ in EMF, given a sub-model EM' of EM derived by the EMF model decomposition algorithm of Section 6.2, we demonstrate the soundness of the EMF model decomposition algorithm by proving the conformance of EM' to the original metamodel $\mathbb{EM}$. More precisely, we will achieve the goal in two steps: Firstly, similar to the sufficient conditions elicited in the abstract setting, i.e., Conditions 1, 2, and 3, we propose two conditions in the EMF setting, i.e., Conditions 4 and 5, and prove that these two conditions will be sufficient to ensure conformance of the EMF sub-model EM'; Secondly, we prove that the sub-model EM' derived by the EMF model decomposition algorithm satisfies Condition 4. As a consequence, if the EMF metamodel $\mathbb{EM}$ satisfies Condition 5, the conformance of EM' to $\mathbb{EM}$ can be concluded. Note that we do not need to add a condition corresponding to Condition 3 in the EMF setting because following the definition of EMF abstraction described above in Section 6.1 and the way source multiplicities are given, all links are fragmentable.

The first sufficient condition (Condition 4 defined below and corresponding to Condition 1) guarantees target reachability in sub-models.

Condition 4 $\forall \mathsf{ra} \in \mathsf{RA}$, $\mathsf{ra_s}(\mathsf{ra}) \in \mathsf{O}'$ *implies* $\mathsf{ra_t}(\mathsf{ra}) \in \mathsf{O}'$.

The second sufficient condition (Condition 5 defined below and corresponding to Condition 2) restricts the nature of the invariants associated to a metamodel to all be forward.

Condition 5 *All invariants of metamodel* $\mathbb{EM}$ *are forward invariants.*

We prove in the following lemma that Conditions 5 and 4 together suffice to ensure the conformance of sub-models in EMF.

Lemma 8 *Given an EMF metamodel* $\mathbb{EM}$ = ($\mathbb{C}$, $\mathbb{DT}$, DV, $\mathbb{Rf}$, $\mathbb{At}$), *an EMF model* EM = ($\mathbb{EM}$, O, τ_{o}, RA, τ_{ra}, $\mathsf{ra_t}$, $\mathsf{ra_s}$, AA, τ_{aa}, $\mathsf{aa_s}$, $\mathsf{aa_t}$), *and an object induced sub-model of* EM, *written* EM' = ($\mathbb{EM}$, O', $\tau_{\mathsf{o}}{}'$, RA', $\tau_{\mathsf{ra}}{}'$, $\mathsf{ra_t}{}'$, $\mathsf{ra_s}{}'$, AA', $\tau_{\mathsf{aa}}{}'$, $\mathsf{aa_s}{}'$, $\mathsf{aa_t}{}'$), *suppose that:*

1. EM *conforms to* $\mathbb{EM}$*;*
2. EM*' satisfies Condition 4;*
3. $\mathbb{EM}$ *satisfies Condition 5;*

then EM*' also conforms to* $\mathbb{EM}$.

Proof See Appendix A.1. □

Moreover, we prove in the following lemma that any sub-model derived by the algorithm satisfies Condition 4.

Lemma 9 *Given an EMF model* EM = ($\mathbb{EM}$, O, τ_{o}, RA, τ_{ra}, $\mathsf{ra_t}$, $\mathsf{ra_s}$, AA, τ_{aa}, $\mathsf{aa_s}$, $\mathsf{aa_t}$) *conforming to an EMF metamodel* $\mathbb{EM}$*, and any object induced sub-model of* EM *derived by the EMF model decomposition algorithm, written* $\mathsf{EM'}$ = ($\mathbb{EM}$, $\mathsf{O'}$, τ_{o}', $\mathsf{RA'}$, τ_{ra}', $\mathsf{ra_t}'$, $\mathsf{ra_s}'$, $\mathsf{AA'}$, τ_{aa}', $\mathsf{aa_s}'$, $\mathsf{aa_t}'$), $\mathsf{EM'}$ *satisfies Condition 4.*

Proof See Appendix A.1. □

Summarizing the discussion above, we thus reach the following conclusion about the soundness of the EMF model decomposition algorithm.

Theorem 10 *Applying our model decomposition technique to a model* EM *conforming to a metamodel* $\mathbb{EM}$ *in EMF following the algorithm of Section 6.2 produces only conformant sub-models if all the invariants of the metamodel are forward.*

Proof Follows from Lemma 8 and Lemma 9. □

7 Application Example: Pruning Based Ecore Model Comprehension

In this section, we demonstrate the power of our generic model decomposition technique by reporting one of its applications in a pruning-based model comprehension method. A typical comprehension question one would like to have answered for a large model is:

> *"Given a set of instances of interest in the model, how does one construct a substantially smaller sub-model that is relevant for the comprehension of these instances?"*

Model readers, when confronted with such a problem, would typically start from the interesting instances and browse through the whole model attempting to manually identify the relevant parts. Even with the best model documentation and the support of model browsing tools, such a task may still be too complicated to solve by hand, especially when the complexity of the original model is high. Moreover, guaranteeing by construction that the identified parts (together with the interesting instances) indeed constitute a valid model further complicates the problem.

Our model decomposition technique can be exploited to provide a linear time automated solution to the problem above. The general idea is to simply take the union of all the scc-nodes, each of which contains at least one interesting instance, and their descendant scc-nodes in the decomposition hierarchy of the original model.

We have implemented the idea in an Ecore model comprehension tool [1] based on the implementation of the EMF model decomposition algorithm defined in Section 6.2. In this context, models to be decomposed are Ecore models (i.e. models conforming to Ecore.ecore). Interestingly Ecore.ecore conforms to itself, in particular to the core part of Ecore as depicted in Figure 5, hence is a metamodel in EMF as defined in Section 4.1.

Listing 1 Constraint model of Ecore implemented in EssentialOCL (part 1).

```
import 'CoreEcore.ecore'

package ecore

context ENamedElement
/*
 * The name can not be empty string
*/
inv WellFormedName:
        self.name.size() > 0

context ETypedElement
/*
 * The lower bound must be greater or equal to 0
*/
inv ValidLowerBound:
        self.lowerBound >= 0
/*
 * The upper bound must be greater than 0 or unlimited (i.e., -1)
*/
inv ValidUpperBound:
        let ub = self.upperBound in
                ub = -1 or ub > 0
/*
 * The lower bound must be less than or equal to the upper bound, unless the upper bound is
    -1
*/
inv ConsistentBounds:
        let lb = self.lowerBound in
        let ub = self.upperBound in
                ub = -1 or lb <= ub
/*
 * Only operations can be without a type to represent void.
*/
inv ValidType:
        self.eType -> isEmpty() implies self.oclIsTypeOf(EOperation)

context EPackage
/*
 * No two sub-packages can have the same name.
*/
inv UniqueSubpackageNames:
        self.eSubpackages -> isUnique(name)
/*
 * No two classifiers can have the same name.
*/
inv UniqueClassifierNames:
        self.eClassifiers -> isUnique(name)

context EOperation
/*
 * No two parameters can have the same name.
*/
inv UniqueParameterNames:
        self.eParameters -> isUnique(name)
/*
 * An operation without a type, which represents void, must have an upper bound of 1.
*/
inv NoRepeatingVoid:
        self.eType -> isEmpty() implies self.upperBound = 1
/*
 * The types of the exceptions raised by an operation are limited to be classes.
*/
inv ValidException:
        self.eExceptions -> forAll(oclIsTypeOf(EClass))

context EClass
def: allFeatures() : Set(EStructuralFeature) =
        self.eSuperTypes -> iterate(
                s: EClass;
                r: Set(EStructuralFeature) = self.eStructuralFeatures ->asSet() |
                r -> union(s.allFeatures())
)
def: allOperations() : Set(EOperation) =
        self.eSuperTypes -> iterate(
```

Listing 2 Constraint model of Ecore implemented in EssentialOCL (part 2).

```
                s: EClass;
                r: Set(EOperation) = self.eOperations -> asSet() |
                r -> union(s.allOperations())
)
def: allSuperTypes(dejaCalled: Set(EClass)) : Set(EClass) =
        self.eSuperTypes -> iterate(
                s: EClass;
                r: Set(EClass) = self.eSuperTypes -> asSet() |
                if dejaCalled->includes(s)
                then r
                else r->union(s.allSuperTypes(dejaCalled->including(s)
                ))
                endif
)
/*
 * No two features (attributes or references) can be the same name.
*/
inv UniqueFeatureNames:
        self.allFeatures() -> isUnique(name)
/*
 * No two operations can have the same signature, which is the name of the operation, the
     number of parameters and their types.
*/
inv UniqueOperationSignatures:
        self.allOperations() -> iterate(
                        op1: EOperation;
                        r1: Boolean = true |
                        r1 and (self.allOperations() -> iterate(
                                op2: EOperation;
                                r2: Boolean = true |
                                r2 and (op1 = op2
                                        or op1.name <> op2.name
                                        or let op1Parameters = op1.eParameters in
                                                let     op2Parameters = op2.eParameters in
                                                op1Parameters -> size() <> op2Parameters ->
                                                    size()
                                                or op1Parameters.eType <> op2Parameters.eType
                                )
                        ))
        )
/*
 * A class can not be a super type of itself.
*/
inv NoCircularSuperTypes:
        self.allSuperTypes(Set{self}) -> excludes(self)

context EEnum
/*
 * No two literals can have the same name.
*/
inv UniqueEnumeratorNames:
        self.eLiterals -> isUnique(name)

context EAttribute
/*
 * The type of an attribute must be a data type
*/
inv ValidType:
        self.eType.oclIsTypeOf(EDataType)

context EReference::container: EBoolean
derive: if eOpposite.oclIsUndefined()
        then false
        else eOpposite.containment
        endif

context EReference
/*
 * The type of a reference must be a class.
*/
inv ValidType:
        self.eType.oclIsTypeOf(EClass)
/*
 * If opposite exists,
 *** it must be a feature of this references's type,
 *** it must have this reference as its opposite,
```

Listing 3 Constraint model of Ecore implemented in EssentialOCL (part 3).

```
 *** this reference and its opposite can not both be containments
*/
inv ConsistentOpposite:
        (self.eOpposite -> isEmpty())
        or (let oppoRef = self.eOpposite in
                        oppoRef.eContainingClass = self.eType
                        and oppoRef.eOpposite = self
                        and (self.containment implies not oppoRef.containment)
        )
/*
 * A container reference (i.e., whose opposite is containment) must have upperbound = 1
*/
inv SingleContainer:
        self.eOpposite -> isEmpty()
        or self.containment implies self.eOpposite.upperBound = 1
/*
 * A reference that may have multiple values must be unique if it is containment or has
     opposite
*/
inv ConsistentUnique:
        ((self.upperBound = -1 or (self.upperBound - self.lowerBound > 1))
        and (self.containment or not self.eOpposite -> isEmpty()))
        implies self.unique
/*
 * If a container reference is required (i.e., lower bound >=1), the contained class can not
     have other container references
*/
inv ConsistentContainer:
        (self.container and self.lowerBound >=1) implies
          self.eContainingClass.allFeatures() -> forAll (f |
                f.oclIsTypeOf(EReference) implies not f.oclAsType(EReference).container
          )

endpackage
```

Consequently, Ecore models are all EMF models in the sense of Section 4.2. Briefly speaking, the tool takes as input an Ecore model and a set of objects in the model marked as interesting for a comprehension task. It runs the EMF model decomposition algorithm, where in the third step, the minimal sub-model in the sub-model lattice which contains all the instances corresponding to a marked object is selected, and returns the result.

7.1 Ecore invariants are all forward

Invariants pertinent to Ecore are identified in Ecore.ecore as annotations nested in the context classes and directly implemented in EcoreValidator.java. For example, the following snippet extracted from Ecore.ecore summarizes the invariants of class ETypedElement, with the following names: ValidLowerBound, ValidUpperBound, ConsistentBounds, and ValidType.

```
<eClassifiers xsi:type="ecore:EClass" name="
    ETypedElement" abstract="true"
    eSuperTypes="#//ENamedElement">
    <eAnnotations source="http://www.eclipse.
        org/emf/2002/Ecore">
      <details key="constraints" value="
          ValidLowerBound ValidUpperBound
          ConsistentBounds ValidType"/>
    </eAnnotations>
 ......
</eClassifiers>
```

We re-implemented the constraint model of Ecore in EssentialOCL. Please refer to Listings 1, 2, and 3 for the details of invariant specifications (started with keyword `inv`) and helper operation specifications (started with keyword `def`) in the constraint model, and to Section 5 for the formal counterpart of EssentialOCL. Thanks to

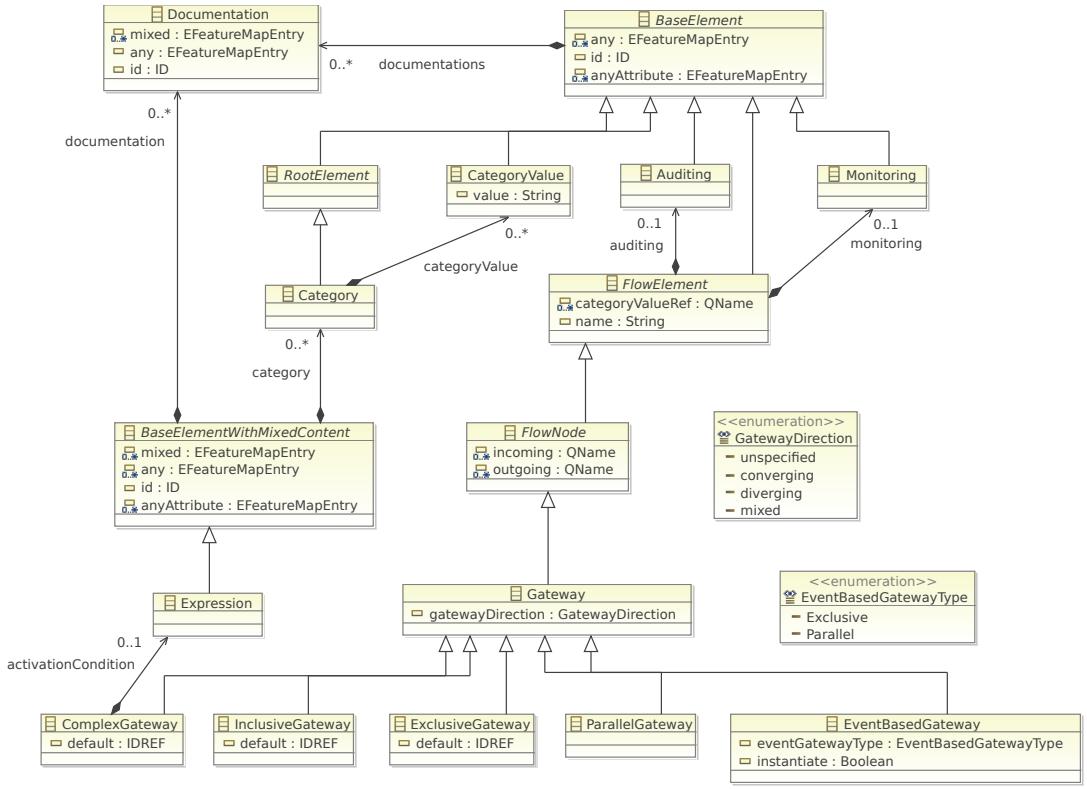

Fig. 7 Pruned class diagram for understanding the Gateway concept in BPMN.

the formal treatment of EssentialOCL in terms of Core-OCL, a precise and formal assessment of the forwardness of the Ecore invariants becomes straightforward, i.e., by checking if allInstances is called in the body of the invariant expression.

We notice that none of the Ecore invariants in Listings 1, 2, 3 makes use of allInstances. Therefore, according to Theorem 6, they are all forward invariants. As a consequence of this fact and by Theorem 10, applying the EMF model decomposition technique to models conforming to Ecore following the steps discussed in Section 6.2 only returns conformant sub-models.

7.2 Case study 1: BPMN

As the first case study we have chosen to comprehend the *Gateway* concept in BPMN [26]: business process model and notation. Gateways are modeling elements in BPMN used to control how sequence flows interact as they converge and diverge within a business process. Five types of gateways are identified in order to cater to different types of sequence flow control semantics: exclusive, inclusive, parallel, complex, and event-based.

Inputs to the comprehension tool for the case study are the following:

– The BPMN Ecore model (bpmn.ecore) containing 156 classifiers (of which 134 are objects of EClass, 11 are objects of EEnum, and 11 are objects of EDataType),

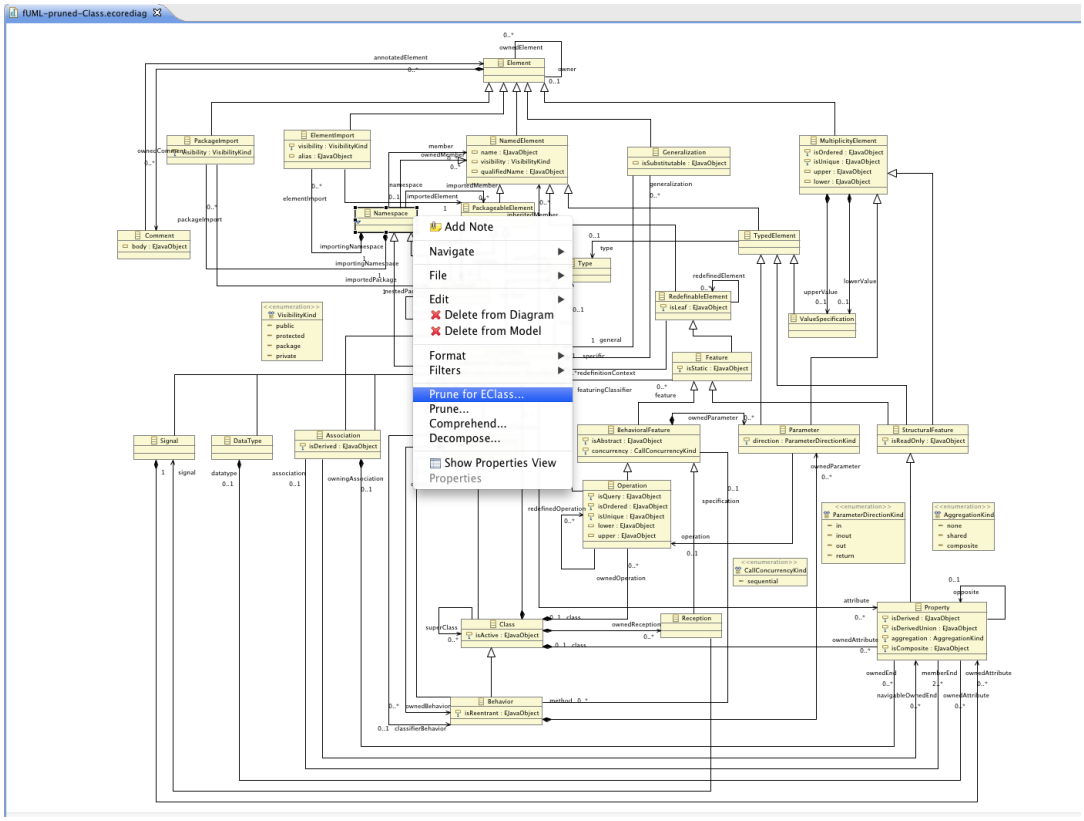

Fig. 8 Launch the comprehension tool interactively from a class diagram view.

252 references (objects of EReference), and 220 attributes (objects of EAttribute). Altogether, it results in a very large class diagram that does not fit on a single page if one wants to be able to read the contents properly.

– a set of EClass objects capturing the key notions of the design of gateways in BPMN: Gateway, ExclusiveGateway, InclusiveGateway, ParallelGateway, ComplexGateway, and EventBasedGateway.

For this input the tool returns in around 0.8 seconds a sub-model of the BPMN Ecore model that conforms to Ecore, contains all the selected interesting objects, and has only 17 classes, 2 enumerations, 7 references, and 21 attributes. We observe that all the other independent concepts of BPMN such as *Activity*, *Event*, *Connector*, and *Artifact*, are pruned out. The class diagram view of the pruned BPMN model is shown in Figure 7. Note that it corresponds well to the class diagram that is sketched in the chapter for describing gateways in the BPMN specification [26]. We have also verified that the pruned BPMN model is indeed a valid Ecore model by calling the Ecore Validator in EMF.

7.3 Case study 2: fUML

As the second case study we have chosen to study fUML [25]: a subset of executable UML models. We take two steps. In the first step, the key concept Class is selected as the seed to produce a smaller sub-model of fUML. Inputs to

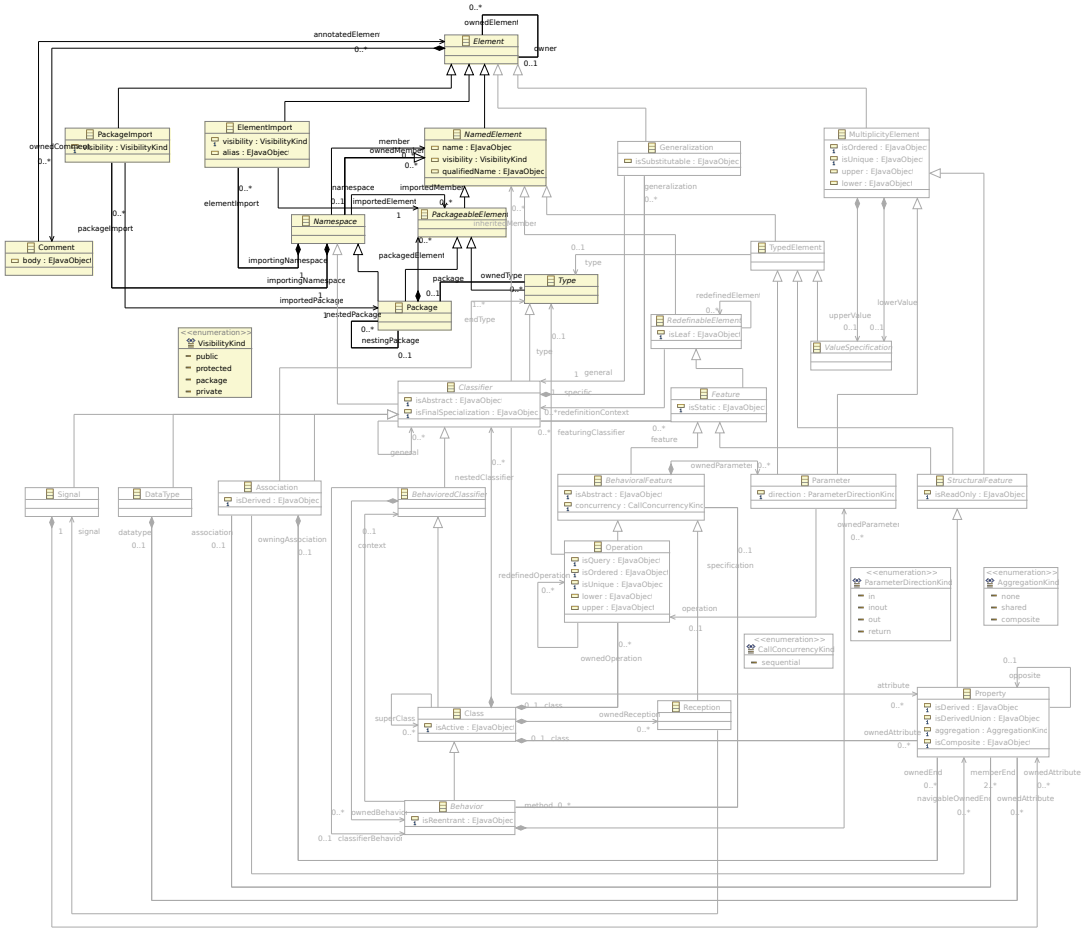

Fig. 9 Pruned class diagram for understanding the Namespace concept in fUML with all irrelevant parts faded out.

the comprehension tool for the first step are the following:

- The fUML Ecore model (fuml.ecore) containing 109 classifiers (in which 104 are objects of EClass and 5 are objects of EEnum), 160 references (objects of EReference), and 56 attributes (objects of EAttribute). Altogether, it adds up to a class diagram that is too large to fit on the screen.
- the EClass object with name Class.

The sub-model returned by the tool contains 28 classes, 4 enumerations, 60 references, and 32 attributes. The computation of the first step takes around 0.04 seconds. Although the number of nodes does not decrease dramatically after this step, the result becomes a class diagram that fits on the screen (see Figure 8).

We then call the comprehension tool one more time on the concept Namespace. Figure 8 illustrates how the tool allows a user to launch the pruning functionality interactively from the Sample Ecore Model Editor in EMF, and Figure 9 shows the result where all irrelevant parts for comprehending namespace are faded out. The highlighted part in Figure 9 contains 9 classes, 1 enumerations, 19 references and 7 attributes, and corresponds well to the class diagram appearing in the fUML specifi-

cation [25] (p. 25). The computation of the second step takes around 0.3 seconds including updating the class diagram view in Sample Ecore Model Editor.

8 Discussion

8.1 About the inclusion conditions

When an instance is included in a sub-model, we decide if a neighbour instance (i.e., an instance connected by either an incoming or an outgoing link) should also be included based on two conditions: Conditions 1 and 3, for the purpose of preserving the conformance of the sub-model. Recall that Condition 1 requires to always follow outgoing links and include also the target instances, and Condition 3 requires to always follow non-fragmentable incoming links and include also the source instances. However, these two conditions are not always necessary. For example, regarding Condition 1, if an outgoing link is *target optional*, i.e., its type has 0 as the lower bound for its target multiplicity, and is not referred to in any invariant evaluation scope, then it is not necessary to also include the target instance to ensure conformance.

The generic nature of our technique and the attempt to avoid sophisticated analysis of invariants account for the existence of such a gap. On one hand, we target a general solution that should work for models expressed in any metamodels. On the other hand, arbitrary invariants can be specified for a metamodel. We opt for a light-weight strategy to automatically guarantee the preservation of invariants in sub-models by simply restricting the invariants inside a property, i.e., being forward, which can be checked easily at the syntax level as proved by Theorem 6. (See Section 8.2 below for discussion on the implication of this restriction.) The evaluation scopes of forward invariants are always reachable from the context instances, and Condition 1 ensures the inclusion of all reachable instances from the context. These two together guarantee a forward invariant always has the same evaluation scope in a sub-model as in the original model, hence preserved.

Egyed reports in [14] an efficient technique to automatically compute evaluation scopes through model profiling. This technique could be combined with our model decomposition technique to reduce the gap from being sufficient to being necessary. More specifically, for all target optional outgoing links from an instance already in a sub-model, the corresponding target instances must be included as well if and only if the latter appear in at least one evaluation scope of the former.

8.2 About allInstances

Theorem 6 allows a user to simply check the occurrence of calls to allInstances in the invariant body in order to decide if an invariant written in CoreOCL (and eventually EssentialOCL) is forward (and eventually to decide if the decomposition technique can be applied). However, the applicability of our model decomposition technique is not limited to only allInstances-free invariants.

Firstly, not all calls to allInstances are necessary. A typical example of abuse is shown below where P stands for some property:

```
context A
inv: A.allInstances() -> forAll(a | P(a))
```

This invariant can be expressed in an equivalent way without using allInstances as follows:

```
context A
inv: P(self)
```

Normalization is a process that removes unnecessary calls to allInstances in a set of invariants and results in an equivalent set of invariants. Two sets of invariants are equivalent if they accept/reject the same set of models. After normalization, the power of forward invariants is the same as the power of invariants excluding allInstances.

Secondly, invariants with necessary calls to allInstances do not alway stop holding after sub-modeling. Actually, a big number of them are *monotonic*, i.e., if an invariant holds for a model then it holds for any sub-models. A typical example of a monotonic invariant checks for identity uniqueness of all instances of a type in a model.

Only in case when invariants with calls to allInstances are both necessary and non-monotonic, the conformance of sub-models is not automatically guaranteed. However, we can still apply our technique but need to re-validate the concerned invariants on the sub-models produced by the decomposition.

We have carried out an empirical study on a pool of 707 invariants written in OCL collected from various sources ranging from OMG (e.g., metamodel specifications of UML, MOF, and OCL), to academic community (e.g., metamodel of RBAC for role-based access control defined at University of Bremen and metamodel of B language defined at IMAG), to industrial community (e.g., the SAM metamodel from the Topcased open source software project). Out of these 707 invariants studied, only 49 call allInstances. And among the 49, 5 can be eliminated by normalization, 39 are monotonic, and only 5 left to be re-validated, which amounts to an overall percentage of 0.7%. Table 2 in the appendix gives a detailed break-down of these numbers.

8.3 EMF opposite links

The model decomposition technique works with associations that have both a source end and a target end. However, in EMF, the notion of a source end is indirectly captured by an opposite reference. This brings consequences to the model decomposition technique. For example, when applying our model decomposition technique in the Ecore model comprehension tool, due to the universal existence of opposite references to all containment references in Ecore (see Figure 5), any Ecore model is a strongly connected graph and consequently, the inclusion of any object in an Ecore model eventually requires the inclusion of the whole model. (Note that it is not a problem caused by containment references themselves, whose links are fragmentable according to our definition in Section 6.1.) A solution to this is to

introduce extra pre-/post-processing before/after model decomposition.

In our Ecore model comprehension application, we omitted two references in the pre-processing: eClassifiers (from EPackage to EClassifier) and eSubPackages (from EPackage to EPackage). This omission breaks the strong bi-directional connection between packages and their members, being either sub-packages or classifiers. It opens up opportunities for a desired degree of decomposition because the inclusion of a member in a package does not imply the inclusion of all the other members in the same package any more (as the outgoing links from packages to their members are omitted). Let us refer to the abridged metamodel by Ecore′. For each Ecore model M, a corresponding Ecore′ model M′ removes from M all links of the two aforementioned references. Then M′ conforms to Ecore′ if M conforms to Ecore. The actual input for the decomposition is Ecore′ (as the metamodel) and M′ (as the model). Note that we have carefully chosen to omit eClassifiers and eSubPackages instead of their opposites, i.e., ePackage and eSuperPackage. Omission of the latter could also bring a good degree of decomposition but would lose the package hierarchy in the result sub-models. Namely, the inclusion of a member does not require the inclusion of its containing package because containment references are fragmentable.

After decomposition, in the post-processing, links of the omitted references can be restored. For each sub-model of M′ in the sub-model lattice, called M′-sub, a corresponding model M-sub adds for each link of ePackage (from EClassifier to EPackage) an opposite link instantiating eClassifiers (from EPackage to EClassifier) and for each link of eSuperPackage (from EPackage to EPackage) an opposite link instantiating eSubPackages (from EPackage to EPackage). Then M-sub conforms to Ecore if M′-sub conforms to Ecore′.

Pre-/post-processings customize our general purpose solution to model decomposition to work in a specific situation and/or for a specific need. They are application specific, and therefore cannot be captured in a general sense but need to be specified by the user case by case. Moreover, extra effort may become necessary to show that the pre-/post-processings do not influence any properties established in the general settings. In the Ecore model comprehension application discussed above, it amounts to demonstrating the following two properties: (1) M′ conforms to Ecore′ if M conforms to Ecore, (2) M-sub conforms to Ecore if M′-sub conforms to Ecore′, which hold straightforwardly by simply checking against the conformance conditions listed in Section 4.3.

9 Related Work

Model slicing In general model slicing consists in identifying sub-models of a model that satisfy certain properties. As such it is a generalization of the work on program slicing to the domain of models. The slicing criterion for model slicing depends on the purpose of the

slicing process. In our case the primary purpose is model comprehension.

Other authors have investigated other uses of model slicing. For instance in [31] the goal is to check satisfiability of a UML class diagram equipped with a set of OCL constraints, i.e., to check the existence of an instance of the class diagram that satisfies all the integrity constraints. Each slice is a valid UML class diagram with constraints belonging to the same slice if they constrain the same model element. The work of [23] strives to establish traceability links between safety requirements and software design elements. The slicing criterion is the inclusion of elements from a SysML model that are relevant for a specific safety requirement.

Another difference between our technique and existing approaches is its genericity: it is not restricted to a particular metamodel. Existing work on model slicing generally applies only to a particular modeling language. Two examples of UML-based model slicing approaches are [20] where model slicing of UML class diagrams is investigated and [5] which considers the problem of slicing the UML metamodel into metamodels corresponding to the different diagram types in UML. With the emergence of an increasing number of domain specific modeling languages genericity becomes an important issue. The need for a generic model slicing technique was identified in [7] which proposed a language for modeling model slicers, allowing for automatic generation of model slicers for any given metamodel.

Note that some work in model slicing allows rewriting of the original model when looking for "sub-models". Such slices are called amorphous slices [4]. These approaches clearly fall outside the scope of our paper since they do not necessarily produce a proper sub-model of the original model.

Metamodel pruning In a similar line of work some authors have investigated the possibility of pruning metamodels in order to make them more manageable. The idea is to remove elements from a metamodel to obtain a minimal set of modeling elements containing a given subset of elements of interest. Such an approach is described in [30]. This work differs from our work in several respects: first, just like model slicing it focuses on a single model rather than considering the collection of relevant sub-models in its totality; second, it is less generic in the sense that it restricts its attention to Ecore metamodels (and the pruning algorithm they present is very dependent on the structure of Ecore), and lastly their goal is not just to get a conformant sub-model but rather to find a sub-metamodel that is a supertype of the original model. This added constraint is due to the main use of the sub-metamodel in model transformation testing.

Model abstraction The general idea of simplifying models (which can be seen as a generalization of model slicing and pruning) has also been investigated in the area of model abstraction (see [18] for an overview). In the area of simulation model abstraction is a method for reducing

the complexity of a simulation model while maintaining the validity of the simulation results with respect to the question that the simulation is being used to address. Work in this area differs from ours in two ways: first, model abstraction techniques generally transform models and do not necessarily result in sub-models; second, conformance of the resulting model with a metamodel is not the main concern but rather validity of simulation results.

Constraint evaluation scope A profiling based technique was implemented in [14] for the computation of evaluation scopes of invariants, where a model profiler monitors what model elements the invariant evaluation engine accesses and logs the accessed model elements in a scope database. Treating all invariants as black boxes, the profiling based technique is independent of the language in which the invariants are expressed. Complementarily, we demonstrate how scopes are computed formally when invariants are all white boxes, i.e., both the syntactic representations and the formal semantics of invariant evaluation are accessible.

In [14], scopes were used to detecting inconsistencies (i.e., breaking of invariants) that occurs during model changes. In our context, we exploit scopes of invariant evaluation to determine if a sub-model automatically preserves invariants satisfied by the original model. More specifically, Theorem 5 demonstrates that letting a sub-model contain the scopes of invariant evaluations suffices to preserve the invariants hold in the original model.

Metamodel/model formalization There has been many attempts at full formalization of metamodeling/modeling. Here we list a few representatives: [11] formally defined the Meta-Modeling Language (a core of UML 2.0) based on the ς-calculus of Cardelli and Abadi; [19] and [6] followed a graph-based approach to formalizing metamodels/models; [2] provided a rich set-theoretic setting for metamodeling; and [8] presented an algebraic semantics of MOF using membership equational logic and term rewriting as a formal foundation.

Our formalization of metamodels and models in Section 2 can be considered as a simplified version of graph-based formalization exploited by work on model transformations such as [6], where a metamodel is expressed as an attributed type graph with inheritance, composition, and multiplicities and a model is captured as a graph typed by the corresponding type graph. In comparison, we omit attributes, containment relations, and operations in metamodels.

Formalizing OCL CoreOCL provides a formal account of EssentialOCL to allow detailed elaborations of evaluation semantics and evaluation scopes. There are various work in the literature formalizing OCL from different aspects: A set theory based denotational semantics for OCL was first defined in [28] then extended in [29]. A type inference system and a big-step operational se-

mantics for OCL 1.4 were defined in [10]. [17] and [15] completed existing work by formalizing the semantics of OCL messages. In [16] a formal semantics is provided for state-based temporal OCL expressions. A graph-based semantics for OCL was developed by translating OCL constraints into expressions over graph rules in [9], and [13] provided a formal semantics of OCL by defining a mapping from OCL to a temporal logic.

10 Conclusion and Future Work

A generic model decomposition technique is put forward in this paper to work with metamodels and models that can be abstracted into a simplified graph-based formalism. A formal foundation is established to demonstrate properties of the technique including correctness and conformance of derived sub-models. We position this generic model decomposition technique as an infrustructural service that can be exploited in many concrete application domains in model-driven software development. A detailed instantiation of the technique to EMF and EssentialOCL and an application of the instantiated model decomposition technique for Ecore model comprehension are presented in this paper. We provide tool implementation and report the results of two concrete case studies on BPMN and fUML comprehension.

We plan to validate the Ecore model comprehension method with more examples from model zoos such as the Repository for Model Driven Development (ReMoDD) [5] and the AtlanMod metamodel zoos [6] to have a better idea of the applicability of the method and the efficiency of the tool implementation, and to discover potential limitations.

Beyond the domain of model comprehension described in this paper, the generic model decomposition technique, being a fundamental facilitating technique, can have more applications in other kinds of tasks in the life cycle of model-driven software development. We mention in the following some examples. (1) Given a software that takes models conforming to a metamodel as input (e.g., a model transformation), an important part in testing the software is test case generation. Our model decomposition technique could help with the generation of new test cases by using one existing test case as the seed. New test cases of various complexity degree could be automatically generated following the sub-model lattice of the seed test case. (2) Our model decomposition could also help with the debugging activity when a failure of the software is observed on a test case. The idea is that we will find a sub-model of the original test case which is responsible for triggering the bug. Although both the reduced test case and the original one are relevant, the smaller test case is easier to understand and investigate. (3) A major obstacle to the massive model reuse in model-based software engineering is the cost of building a repository of reusable model components. A more effective alternative to creating those reusable model com-

[5] http://www.cs.colostate.edu/remodd/v1/

[6] http://www.emn.fr/z-info/atlanmod/index.php/Zoos

ponents from scratch is to discover them from existing system models. Sub-models of a system extracted by following our model decomposition technique are all guaranteed to be valid models hence can be wrapped up into modules and reused in the construction of other systems following our modular model composition paradigm [21]. (4) Finally, applying the model decomposition technique to system models in multi-view modeling so that distinct and separate sub-models capturing different aspects of a system can be extracted is a potential avenue for future work.

Implementation at this stage is only for proof of concept. As a consequence, neither user-friendliness nor efficiency is of high priority in the current version (although the performance is still agreeable, i.e., within a second for the case studies). Optimization is also in our future work agenda. We plan to equipe the comprehension tool with a better graphical user interface and provide the tool as an Eclipse plugin to reach a wider audience.

References

1. Democles tool. `http://democles.lassy.uni.lu/`.
2. Marcus Alanen and Ivan Porres. A metamodeling language supporting subset and union properties. *Software and System Modeling (SoSyM)*, 7(1):103–124, 2008.
3. Moussa Amrani. A Formal Semantics of Kermeta. In Marjan Mernik, editor, *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, chapter 10. IGI Global, 2012.
4. Kelly Androutsopoulos, David Clark, Mark Harman, Robert M. Hierons, Zheng Li, and Laurence Tratt. Amorphous slicing of extended finite state machines. *IEEE Transactions on Software Engineering*, 99(PrePrints):1, 2012.
5. Jung Ho Bae, KwangMin Lee, and Heung Seok Chae. Modularization of the UML metamodel using model slicing. *Fifth International Conference on Information Technology: New Generations*, 0:1253–1254, 2008.
6. Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Formal foundation of consistent emf model transformations by algebraic graph transformation. *Software and Systems Modeling (SoSyM)*, 11(2):227–250, 2012.
7. Arnaud Blouin, Benoît Combemale, Benoit Baudry, and Olivier Beaudoux. Modeling model slicers. In *the Proceedings of ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2011)*, pages 62–76, 2011.
8. Artur Boronat and José Meseguer. An algebraic semantics for MOF. *Formal Aspects of Computing*, 22(3-4):269–296, 2010.
9. Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Gabriele Taentzer. Consistency checking and visualization of OCL constraints. In *the Proceedings of 3rd International Conference on The Unified Modeling Language: Advancing the Standard (UML 2001)*, LNCS 1393, pages 294–308, 2000.
10. María Victoria Cengarle and Alexander Knapp. A formal semantics for OCL 1.4. In *the Proceedings of 4th International Conference on the Unified Modeling Language: Modeling Languages, Concepts, and Tools (UML*

2001), LNCS 2185, pages 118–133, 2001.

11. Tony Clark, Andy Evans, and Stuart Kent. The metamodelling language calculus: Foundation semantics for UML. In *the Proceedings of 4th International Conference on 4th International Conference (FASE 2001)*, LNCS 2029, pages 17–31, 2001.
12. Reinhard Diestel. *Graph Theory Fourth Edition*. Springer-Verlag, 2010.
13. Dino Distefano, Joost-Pieter Katoen, and Arend Rensink. On a temporal logic for object-based systems. In *the Proceedings of 4th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2000)*, pages 305–325, 2000.
14. Alexander Egyed. Automatically detecting and tracking inconsistencies in software design models. *IEEE Transactions on Software Engineering*, 37(2):188–204, 2011.
15. Stephan Flake. Towards the completion of the formal semantics of OCL 2.0. In *the Proceedings of 27th Australasian Conference on Computer science (ACSC) '04*, pages 73–82, 2004.
16. Stephan Flake and Wolfgang Mueller. Formal semantics of static and temporal state-oriented OCL constraints. *Software and Systems Modeling (SoSyM)*, 2:164–186, 2003.
17. Stephan Flake and Wolfgang Müller. Formal semantics of OCL messages. *Electronic Notes in Theoretical Computer Science*, 102:77–97, 2004.
18. F.K. Frantz. A taxonomy of model abstraction techniques. *Winter Simulation Conference*, 0:1413–1420, 1995.
19. Frédéric Jouault and Jean Bézivin. KM3: A DSL for metamodel specification. In *the Proceedings of 8th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2006)*, LNCS 4037, pages 171–185, 2006.
20. Huzefa Kagdi, Jonathan I. Maletic, and Andrew Sutton. Context-free slicing of UML class models. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 635–638, Washington, DC, USA, 2005. IEEE Computer Society.
21. Pierre Kelsen and Qin Ma. A modular model composition technique. In *the Proceedings of 13th International Conference on Fundamental Approaches to Software Engineering, (FASE 2010)*, volume LNCS 6013, pages 173–187, 2010.
22. Pierre Kelsen, Qin Ma, and Christian Glodt. Models within models: Taming model complexity using the submodel lattice. In *the Proceedings of 14th International Conference on Fundamental Approaches to Software Engineering, (FASE 2011)*, volume LNCS 6603, pages 171–185, 2011.
23. Shiva Nejati, Mehrdad Sabetzadeh, Davide Falessi, Lionel C. Briand, and Thierry Coq. A sysml-based approach to traceability management and design slicing in support of safety certification: Framework, tool support, and case studies. *Information & Software Technology*, 54(6):569–590, 2012.
24. OMG. Meta Object Facility (MOF) Core Specification Version 2.4.1, 2011.
25. OMG. Semantics of a Foundational Subset for Executable UML Models (fUML), v1.0, 2011.
26. OMG. Business Process Model and Notation (BPMN) Version 2.0, 2012.
27. OMG. Object Constraint Language Version 2.3.1, 2012.

28. Mark Richters and Martin Gogolla. On formalizing the UML object constraint language OCL. In *the Proceedings of 17th International Conference on Conceptual Modeling (ER '98)*, pages 449–464, 1998.
29. Mark Richters and Martin Gogolla. OCL: Syntax, semantics, and tools. In *the Proceedings of Object Modeling with the OCL, The Rationale behind the Object Constraint Language (OCL 2002)*, LNCS 2263, pages 42–68, 2002.
30. Sagar Sen, Naouel Moha, Benoit Baudry, and Jean-Marc Jézéquel. Meta-model pruning. In *the Proceedings of the 12th international conference on Model Driven Engineering Languages and Systems (MoDELS 2009)*, pages 32–46, 2009.
31. Asadullah Shaikh, Robert Clarisó, Uffe Kock Wiil, and Nasrullah Memon. Verification-driven slicing of UML/OCL models. In *the Proceedings of 25th IEEE/ACM International Conference on Automated Software Engineering(ASE 2010)*, pages 185–194, 2010.
32. Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework, 2nd Edition.* Addison-Wesley Professional, 2008.
33. Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.
34. Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA.* Addison-Wesley Professional, 2003.

A Appendix

A.1 Proofs

Theorem 2 *Given a model* $\mathsf{M} = (\mathbb{M}, \mathsf{N}, \mathsf{A}, \tau, \mathsf{src}, \mathsf{tgt})$ *and an instance induced sub-model* $\mathsf{M}' = (\mathbb{M}, \mathsf{N}', \mathsf{A}', \tau', \mathsf{src}', \mathsf{tgt}')$ *of* M, M' *satisfies both Conditions 1 and 3 if and only if there exists a corresponding antichain-node of the decomposition hierarchy of* M *where* M' *is induced by all the instances grouped in this antichain-node.*

Proof We first demonstrate that if M' is induced by the set of instances that are grouped in an antichain-node α of the decomposition hierarchy of M, then M' satisfies both Conditions 1 and 3.

- Check M' against Condition 1: following the algorithm in Section 3.2 to compute the decomposition hierarchy, given a link $\mathsf{a} \in \mathsf{A}$, $\mathsf{src}(\mathsf{a})$ and $\mathsf{tgt}(\mathsf{a})$ are either grouped in one scc-node, or the scc-node s_1 that groups $\mathsf{src}(\mathsf{a})$ is different from the scc-node s_2 that groups $\mathsf{tgt}(\mathsf{a})$ and s_2 is a descendant of s_1 because of the presence of a (which must be a fragmentable link). In both cases, grouping $\mathsf{src}(\mathsf{a})$ in an antichain-node implies grouping also $\mathsf{tgt}(\mathsf{a})$ in the same antichain-node. Therefore, if $\mathsf{src}(\mathsf{a}) \in \mathsf{N}'$, i.e., $\mathsf{src}(\mathsf{a})$ is grouped in the antichain-node α, so is $\mathsf{tgt}(\mathsf{a})$, i.e., $\mathsf{tgt}(\mathsf{a}) \in \mathsf{N}'$.
- Check M' against Condition 3: following the algorithm in Section 3.2 to compute the decomposition hierarchy, given a link $\mathsf{a} \in \mathsf{A}$ that is non-fragmentable, $\mathsf{src}(\mathsf{a})$ and $\mathsf{tgt}(\mathsf{a})$ are grouped in one wcc-node, hence are grouped in one scc-node, hence will be grouped always together in any antichain-nodes. Therefore, if $\mathsf{tgt}(\mathsf{a}) \in \mathsf{N}'$, i.e., $\mathsf{tgt}(\mathsf{a})$ is grouped in the antichain-node α, so is $\mathsf{src}(\mathsf{a})$, i.e., $\mathsf{src}(\mathsf{a}) \in \mathsf{N}'$.

We now demonstrate the other direction of the theorem, namely, if M' satisfies both Conditions 1 and 3, then there exists an antichain-node of the decomposition hierarchy of M such that M' is induced by the set of instances that are grouped in this antichain-node.

Let S refer to the set of scc-nodes in the decomposition hierarchy, each of which includes at least one instance of M'.

1. All the instances that are grouped in an scc-node in S belong to M'. Indeed given an scc-node $s \in S$, there must exist an instance n grouped in s and $\mathsf{n} \in \mathsf{N}'$ in order for s to be included in S. Let n' be another instance grouped in s. Following the algorithm in Section 3.2 to compute the decomposition hierarchy, n and n' are grouped in one scc-node either in the first or the second step.
   (a) If they are grouped in the first step, that means the two instances are weakly connected by non-fragmentable links, and because M' satisfies Conditions 1 and 3, n' should also be in M'.
   (b) If they are grouped in the second step but not in the first step, that means n and n' are grouped in two separate wcc-nodes in the first step, called w

and w', which are strongly connected by a path of fragmentable links. Referring to the other wcc-nodes on the path by $w_1, \ldots, w_k$, there exists a set of instances $\mathsf{n}_0 \in w$, $\mathsf{n}'_0 \in w'$, and $\mathsf{n}_i, \mathsf{n}'_i \in w_i$ for $1 \leq i \leq k$, such that the following fragmentable links exist: from n_0 to n_1, from n'_i to n_{i+1} ($\forall i.1 \leq i < k$), and from n'_k to n'_0. Since M' satisfies Condition 1, if the source instance of these fragmentable links belongs to M', so does the target instance. Because the following pairs of instances: n and n_0, n_i and n'_i ($\forall i.1 \leq i \leq k$), and n'_0 and n', are respectively grouped in a wcc-node, if one instance in a pair belongs to M' then the other instance in the pair must belong to M' as well following Conditions 1 and 3. From the above discussion and by applying mathematical induction, n belonging to M' implies that n' belongs to M' as well.

2. S constitutes an antichain plus descendant. We partition S into two subsets: S_1 contains all the scc-nodes in S that do not have another scc-node also in S as ancestor; S_2 contains the rest, i.e., $S_2 = S \setminus S_1$. Clearly S_1 constitutes an antichain, and any scc-node in S_2 is a descendant of an scc-node in S_1 because otherwise the former scc-node should belong to S_1 instead of S_2. Moreover, S_2 contains all the descendants of scc-nodes in S_1. Indeed given a child s_2 of an scc-node $s_1 \in S_1$, the fragmentable link from s_1 to s_2 connects an instance n_1 grouped in s_1 to an instance n_2 grouped in s_2. Because $s_1 \in S_1$, following the demonstrated item 1 above, we have $\mathsf{n}_1 \in \mathsf{N}'$. Because of the outgoing fragmentable link from n_1 to n_2 and since M' satisfies Condition 1, we also have $\mathsf{n}_2 \in \mathsf{N}'$. Therefore we have $s_2 \in S$. Furthermore, $s_2 \notin S_1$ because it has $s_1 \in S$ as its ancestor. Hence we have $s_2 \in S_2$. Inductively we conclude that any descendant of s_1 belongs to S_2 and hence S constitutes an antichain plus descendant.
3. Collapse all the scc-nodes in S into an antichain-node called α. We demonstrate that M' is induced by the instances grouped in α, i.e., N' equals to the set of instances grouped in α: any instance of M' is grouped in α because of the selection criteria of S, and any instance grouped in α is an instance of M' following the demonstrated item 1 above.

□

Lemma 4 *Given a model* EM*, an evaluation environment* Λ*, and an expression of CoreOCL* $\mathbb{e}$*, if* $\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e} \Downarrow (\mathsf{v}, \mathsf{O}_S)$ *holds, then for any object induced sub-model* EM' *of* EM *where* $\mathsf{O}_S \subseteq \mathsf{O}'$*,* $\Lambda \overset{\mathsf{EM}'}{\vDash} \mathbb{e} \Downarrow (\mathsf{v}, \mathsf{O}_S)$ *also holds.*

Proof We prove by induction on the depth of the structure of the expression $\mathbb{e}$.

Base cases: expressions of depth 1

($\mathbb{e} \equiv \mathsf{dv}$) Trivial.

($\mathbb{e} \equiv x$) Because of the same evaluation environment Λ and $\mathsf{O}_S \subseteq \mathsf{O}'$.

($\mathbb{e} \equiv \mathsf{Bag\{\}} \mid \mathsf{Set\{\}} \mid \mathsf{Seq\{\}} \mid \mathsf{OSet\{\}}$) Trivial.

($\mathbb{e} \equiv \mathsf{allInstances}(\mathbb{c})$) Because of $\{\mathsf{o} \mid \mathsf{o} \in \mathsf{O} \text{ and } \tau_{\mathsf{o}}(\mathsf{o}) \overset{\mathsf{EM}}{=} \mathbb{c}\} \subseteq \mathsf{O}'$, and the fact that EM' is an object induced sub-model of EM, we have $\{\mathsf{o} \mid \mathsf{o} \in \mathsf{O}' \text{ and } \tau_{\mathsf{o}}'(\mathsf{o}) \overset{\mathsf{EM}'}{=} \mathbb{c}\} = \{\mathsf{o} \mid \mathsf{o} \in \mathsf{O} \text{ and } \tau_{\mathsf{o}}(\mathsf{o}) \overset{\mathsf{EM}}{=} \mathbb{c}\}$.

Induction cases: expressions of depth n We suppose that for any expression of depth $< n$, the lemma holds. We prove that it is also the case for expressions of depth n. We distinguish the top most structure the expression may have:

($\mathbb{e} \equiv \mathbb{e}_1.\mathbb{sf}$) Assume $\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}_1.\mathbb{sf} \Downarrow (\mathsf{v}, \mathsf{O}_S)$ holds. Following StructuralFeatureCall, we have both $\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}_1 \Downarrow (\mathsf{o}, \mathsf{O}_S^1)$ and $\mathsf{o}.\mathbb{sf} \overset{\mathsf{EM}}{=} \mathsf{v}$ hold, where $\mathsf{O}_S = \mathsf{O}_S^1 \cup \mathsf{O}_S(\mathsf{v})$. By induction hypothesis, we have also $\Lambda \overset{\mathsf{EM}'}{\vDash} \mathbb{e}_1 \Downarrow (\mathsf{o}, \mathsf{O}_S^1)$ holds. According to Lemma 3 and the lemma assumption, $\mathsf{o} \in \mathsf{O}_S^1 \subseteq \mathsf{O}_S \subseteq \mathsf{O}'$ and $\mathsf{O}_S(\mathsf{v}) \subseteq \mathsf{O}_S \subseteq \mathsf{O}'$. Because EM' is an object induced sub-model of EM, $\mathsf{o}.\mathbb{sf}$ computes to the same value v in EM' as in EM according to Table 1. Namely, we also have $\mathsf{o}.\mathbb{sf} \overset{\mathsf{EM}'}{=} \mathsf{v}$. As a consequence, by applying rule StructuralFeatureCall, we have $\Lambda \overset{\mathsf{EM}'}{\vDash} \mathbb{e}_1.\mathbb{sf} \Downarrow (\mathsf{v}, \mathsf{O}_S^1 \cup \mathsf{O}_S(\mathsf{v}))$ holds, i.e., $\Lambda \overset{\mathsf{EM}'}{\vDash} \mathbb{e} \Downarrow (\mathsf{v}, \mathsf{O}_S)$ holds.

($\mathbb{e} \equiv \mathsf{Tuple}\{\overline{\mathbb{tn}} : \overline{\mathbb{t}} = \overline{\mathsf{v}}\}$) Following Tuple and by induction on the sub-expressions $\overline{\mathbb{e}}$.

($\mathbb{e} \equiv \mathbb{e}_1.\mathbb{tn}$) Following TuplePartCall and by induction on the sub-expression $\mathbb{e}_1$.

($\mathbb{e} \equiv \mathsf{let}\ x = \mathbb{e}_1\ \mathsf{in}\ \mathbb{e}_2$) Following LetBinding and by induction on the two sub-expressions $\mathbb{e}_1$ and $\mathbb{e}_2$.

($\mathbb{e} \equiv \mathsf{def}\ \mathbb{c} :: \mathbb{op} = \lambda(\overline{x} : \overline{\mathbb{t}}).\mathbb{e}_1\ \mathsf{in}\ \mathbb{e}_2$) Following DefExp and by induction on the sub-expression $\mathbb{e}_2$.

($\mathbb{e} \equiv \mathbb{e}_1.\mathbb{op}(\overline{x} := \overline{\mathbb{e}})$) Following ObjectOperationCall and by induction on the sub-expressions: $\mathbb{e}_1$, $\overline{\mathbb{e}}$, and body expression of the operation $\mathbb{e}_{\mathbb{op}}$.

($\mathbb{e} \equiv \kappa(\overline{\mathbb{e}})$) Following KappaCall and by induction on the sub-expressions $\overline{\mathbb{e}}$.

($\mathbb{e} \equiv \mathsf{if}\ \mathbb{e}_1\ \mathsf{then}\ \mathbb{e}_2\ \mathsf{else}\ \mathbb{e}_3$) Following IfTrue (respectively IfFalse) and by induction on the sub-expressions $\mathbb{e}_1$ and $\mathbb{e}_2$ (respectively $\mathbb{e}_3$).

($\mathbb{e} \equiv \mathbb{e}_1\ \mathsf{asInstanceOf}\ \mathbb{c}$) Following DownCastOK (respectively DownCastNotOK), by induction on the sub-expression $\mathbb{e}_1$, and because EM' is an object induced sub-model of EM, i.e., for an object o in both EM and EM', $\tau_{\mathsf{o}}(\mathsf{o}) \preccurlyeq \mathbb{c}$ iff $\tau_{\mathsf{o}}'(\mathsf{o}) \preccurlyeq [\mathsf{EM}']\mathbb{c}$ (respectively $\tau_{\mathsf{o}}(\mathsf{o}) \not\preccurlyeq \mathbb{c}$ iff $\tau_{\mathsf{o}}'(\mathsf{o}) \not\preccurlyeq [\mathsf{EM}']\mathbb{c}$).

($\mathbb{e} \equiv \mathbb{e}_1\ \mathsf{isInstanceOf}\ \mathbb{c}$) Following IsInstanceOfTrue (respectively IsInstanceOfFalse), by induction on the sub-expression $\mathbb{e}_1$, and because EM' is an object induced sub-model of EM, i.e., for an object o in both EM and EM', $\tau_{\mathsf{o}}(\mathsf{o}) \overset{\mathsf{EM}}{=} \mathbb{c}$ iff $\tau_{\mathsf{o}}'(\mathsf{o}) \overset{\mathsf{EM}'}{=} \mathbb{c}$.

($\mathbb{e} \equiv \mathbb{e}_1\ \mathsf{isOfKind}\ \mathbb{c}$) Following IsKindOfTrue (respectively IsKindOfFalse), by induction on the sub-expression $\mathbb{e}_1$, and because EM' is an object induced sub-model of EM, i.e., for an object o in both EM and EM', $\tau_{\mathsf{o}}(\mathsf{o}) \overset{\mathsf{EM}}{=} \mathbb{c}'$ iff $\tau_{\mathsf{o}}'(\mathsf{o}) \overset{\mathsf{EM}'}{=} \mathbb{c}'$; and EM and EM' have the same metamodel where $\mathbb{c}' \preccurlyeq \mathbb{c}$ (respectively $\mathbb{c}' \not\preccurlyeq \mathbb{c}$).

($\mathbb{e} \equiv \mathbb{e}_1 \rightarrow \mathsf{iterate}(x; y = \mathbb{e}_2 \mid \mathbb{e}_3)$) Following evaluation rule CollectionIteration and by induction on the sub-expressions $\mathbb{e}_1$, $\mathbb{e}_2$ and $\mathbb{e}_3$.

□

Theorem 6 *Without calling* allInstances*, CoreOCL invariants are all forward following Definition 8.*

Proof Let $\mathbb{einv} = (\mathbb{c}, \mathbb{e})$ be a CoreOCL invariant defined for the metamodel of a model EM where $\mathbb{e}$ is an expression written in the sub-language of CoreOCL excluding allInstances. Let o be an object of EM where $\tau_{\mathsf{o}}(\mathsf{o}) \preccurlyeq \mathbb{c}$. Let Λ be the initial evaluation environment for $\mathsf{eval}(\mathbb{einv}, \mathsf{EM}, \mathsf{o})$. Suppose $\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e} \Downarrow (\mathsf{v}, \mathsf{O}_S)$. The set of objects in the scope of $(\mathbb{einv}, \mathsf{EM}, \mathsf{o})$, i.e., O_S, is reachable from o following the first statement of Lemma 7. □

Lemma 7 *Given a model* EM*, an object* o *in it, a CoreOCL expression* $\mathbb{e}$ *with no calls to* allInstances *(i.e.,* $\mathbb{e}$ *is an expression written in the sub-language of CoreOCL excluding* allInstances*), and an evaluation environment* Λ *in which values bound to variables only refer to objects reachable from* o *if any, i.e.,* $\forall x$ *in the domain of* Λ*,* $\forall \mathsf{o}' \in \mathsf{O}_S(\Lambda(x))$*,* o' *is reachable from* o *(see Definition 7). Suppose* $\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e} \Downarrow (\mathsf{v}, \mathsf{O}_S)$*. We have the following two statements hold:*

1. $\forall \mathsf{o}' \in \mathsf{O}_S$*,* o' *is reachable from* o*;*
2. $\forall \mathsf{o}' \in \mathsf{O}_S(\mathsf{v})$*,* o' *is reachable from* o*.*

Proof We prove by induction on the depth of the structure of the expression $\mathbb{e}$.

Base cases: expressions of depth 1

($\mathbb{e} \equiv \mathsf{dv}$) Trivial.

($\mathbb{e} \equiv x$) Because values bound in the evaluation environment Λ only refer to objects reachable from o.

($\mathbb{e} \equiv \mathsf{Bag\{\}} \mid \mathsf{Set\{\}} \mid \mathsf{Seq\{\}} \mid \mathsf{OSet\{\}}$) Trivial.

Induction cases: expressions of depth n We suppose that for any expression of depth $< n$, the two statements hold. We prove that it is also the case for expressions of depth n. We distinguish the top most structure the expression may have:

($\mathbb{e} \equiv \mathbb{e}_1.\mathbb{sf}$) Assume $\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}_1.\mathbb{sf} \Downarrow (\mathsf{v}, \mathsf{O}_S)$ holds. Following StructuralFeatureCall, we have $\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}_1 \Downarrow (\mathsf{o}_1, \mathsf{O}_S^1)$ and $\mathsf{o}_1.\mathbb{sf} \overset{\mathsf{EM}}{=} \mathsf{v}$ hold, where $\mathsf{O}_S = \mathsf{O}_S^1 \cup \mathsf{O}_S(\mathsf{v})$. By induction hypothesis, we have o_1 reachable from o. Therefore we have also objects in $\mathsf{O}_S(\mathsf{v})$ all reachable from o via o_1 and $\mathbb{sf}$ following Table 1 and Definition 7. As a consequence, objects in O_S are all reachable from o because it is the case for all the objects in O_S^1 (by induction hypothesis) and $\mathsf{O}_S(\mathsf{v})$.

($\mathbb{e} \equiv \mathsf{Tuple}\{\overline{\mathbb{tn} : \mathbb{t} = \mathbb{e}}\}$) Assume $\Lambda \overset{\mathsf{EM}}{\vDash} \mathsf{Tuple}\{\overline{\mathbb{tn} : \mathbb{t} = \mathbb{e}}\} \Downarrow (\mathsf{v}, \mathsf{O}_S)$ holds. Following Tuple, we have $\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}_i \Downarrow (\mathsf{v}_i, \mathsf{O}_S^i)$ holds for all $\mathbb{e}_i \in \overline{\mathbb{e}}$ where $1 \leq i \leq k$, $\mathsf{v} = \mathsf{Tuple}\{\mathbb{tn}_1 : \mathbb{t}_1 = \mathsf{v}_1, \ldots, \mathbb{tn}_k : \mathbb{t}_k = \mathsf{v}_k\}$, and $\mathsf{O}_S = \bigcup_{1 \leq i \leq k} \mathsf{O}_S^i$. By induction hypothesis, objects in O_S^i and $\mathsf{O}_S(\mathsf{v}_i)$ are all reachable from o for all $1 \leq i \leq k$. Therefore, objects in O_S and objects in $\mathsf{O}_S(\mathsf{v}) = \mathsf{O}_S(\mathsf{Tuple}\{\mathbb{tn}_1 : \mathbb{t}_1 = \mathsf{v}_1, \ldots, \mathbb{tn}_k : \mathbb{t}_k = \mathsf{v}_k\}) = \bigcup_{1 \leq i \leq k} \mathsf{O}_S(\mathsf{v}_i)$ are all be reachable from o.

($\mathbb{e} \equiv \mathbb{e}_1.\mathbb{t}\mathbb{n}$) Assume $\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}_1.\mathbb{t}\mathbb{n} \Downarrow (\mathsf{v}, \mathsf{O}_S)$ holds. Following TuplePartCall, we have $\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}_1 \Downarrow (\mathsf{Tuple}\{\overline{\mathbb{t}\mathbb{n}} : \bar{\mathbb{t}} = \bar{\mathsf{v}}\}, \mathsf{O}_S)$ holds. By induction, objects in O_S and $\mathsf{O}_S(\mathsf{Tuple}\{\overline{\mathbb{t}\mathbb{n}} : \bar{\mathbb{t}} = \bar{\mathsf{v}}\}) = \bigcup_{\mathsf{v}_i \in \bar{\mathsf{v}}} \mathsf{O}_S(\mathsf{v}_i)$ are all reachable from o. Therefore, objects in $\mathsf{O}_S(\mathsf{v}_i)$ are all reachable from o.

($\mathbb{e} \equiv \mathsf{let}\ x = \mathbb{e}_1\ \mathsf{in}\ \mathbb{e}_2$) Following LetBinding, and by induction on the two sub-expressions $\mathbb{e}_1$ and $\mathbb{e}_2$.

($\mathbb{e} \equiv \mathsf{def}\ \mathbb{c} :: \mathbb{o}\mathbb{p} = \lambda(\overline{x} : \bar{\mathbb{t}}).\mathbb{e}_1\ \mathsf{in}\ \mathbb{e}_2$) Following DefExp and by induction on the sub-expression $\mathbb{e}_2$.

($\mathbb{e} \equiv \mathbb{e}_1.\mathbb{o}\mathbb{p}(\overline{x} := \bar{\mathbb{e}})$) Assume $\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}_1.\mathbb{o}\mathbb{p}(\overline{x} := \bar{\mathbb{e}}) \Downarrow (\mathsf{v}, \mathsf{O}_S)$. Following ObjectOperationCall, we have $\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}_1 \Downarrow (\mathsf{o}_1, \mathsf{O}_S^{\mathsf{o}_1})$, $\Lambda \overset{\mathsf{EM}}{\vDash} \bar{\mathbb{e}} \Downarrow (\bar{\mathsf{v}}, \overline{\mathsf{O}_S})$, and $\Lambda \oplus \{(\mathsf{self} : \mathsf{o}_1)\} \oplus \{(\overline{x} : \bar{\mathsf{v}})\} \overset{\mathsf{EM}}{\vDash} \mathbb{e}_{\mathbb{o}\mathbb{p}} \Downarrow (\mathsf{v}, \mathsf{O}_S^{\mathbb{o}\mathbb{p}})$, where $\mathsf{O}_S = \mathsf{O}_S^{\mathsf{o}_1} \cup (\bigcup \overline{\mathsf{O}_S}) \cup \mathsf{O}_S^{\mathbb{o}\mathbb{p}}$. By induction hypothesis, we have o_1, objects in $\mathsf{O}_S^{\mathsf{o}_1}$, objects in $\mathsf{O}_S(\bar{\mathsf{v}})$, and objects in $\overline{\mathsf{O}_S}$ all reachable from o. Therefore, the overridden environment $\Lambda \oplus \{(\mathsf{self} : \mathsf{o}_1)\} \oplus \{(\overline{x} : \bar{\mathsf{v}})\}$ binds variables to values that only refer to objects reachable from o if any. Hence by induction hypothesis, objects in $\mathsf{O}_S(\mathsf{v})$ and $\mathsf{O}_S^{\mathbb{o}\mathbb{p}}$ are all reachable from o. As a consequence, objects in O_S are all reachable from o.

($\mathbb{e} \equiv \kappa(\mathbb{e}_1, \ldots, \mathbb{e}_k)$) Assume $\Lambda \overset{\mathsf{EM}}{\vDash} \kappa(\mathbb{e}_1, \ldots, \mathbb{e}_k) \Downarrow (\mathsf{v}, \mathsf{O}_S)$ holds. Following KappaCall, we have $\Lambda \overset{\mathsf{EM}}{\vDash} \mathbb{e}_i \Downarrow (\mathsf{v}_i, \mathsf{O}_S^i)$ holds for $1 \leq i \leq k$, $\kappa(\mathsf{v}_1, \ldots, \mathsf{v}_k) = \mathsf{v}$, and $\mathsf{O}_S = \bigcup_{1 \leq i \leq k} \mathsf{O}_S^i$. By induction hypothesis, objects in O_S^i and $\mathsf{O}_S(\mathsf{v}_i)$ are all reachable from o for $1 \leq i \leq k$. Therefore, objects in O_S are all reachable from o. Moreover, κ being the pre-defined operations, does not introduce any new objects other than those are referenced in its operands, i.e., v_i for $1 \leq i \leq k$. Hence objects in $\mathsf{O}_S(\mathsf{v})$ should all be reachable from o as well.

($\mathbb{e} \equiv \mathsf{if}\ \mathbb{e}_1\ \mathsf{then}\ \mathbb{e}_2\ \mathsf{else}\ \mathbb{e}_3$) Following IfTrue (respectively IfFalse) and by induction on the sub-expressions $\mathbb{e}_1$ and $\mathbb{e}_2$ (respectively $\mathbb{e}_3$).

($\mathbb{e} \equiv \mathbb{e}_1\ \mathsf{asInstanceOf}\ \mathbb{c}$) Following DownCastOK (respectively DownCastNotOK) and by induction on the sub-expression $\mathbb{e}_1$.

($\mathbb{e} \equiv \mathbb{e}_1\ \mathsf{isInstanceOf}\ \mathbb{c}$) Following IsInstanceOfTrue (respectively IsInstanceOfFalse) and by induction on the sub-expression $\mathbb{e}_1$.

($\mathbb{e} \equiv \mathbb{e}_1\ \mathsf{isOfKind}\ \mathbb{c}$) Following IsKindOfTrue (respectively IsKindOfFalse) and by induction on the sub-expression $\mathbb{e}_1$.

($\mathbb{e} \equiv \mathbb{e}_1 \rightarrow \mathsf{iterate}(x; y = \mathbb{e}_2 \,|\, \mathbb{e}_3)$) Following evaluation rule CollectionIteration and by induction on the sub-expressions $\mathbb{e}_1$, $\mathbb{e}_2$ and $\mathbb{e}_3$.

□

Lemma 8 *Given an EMF metamodel* $\mathbb{EM} = (\mathbb{C}, \mathbb{DT}, \mathsf{DV}, \mathbb{Rf}, \mathbb{At})$, *an EMF model* $\mathsf{EM} = (\mathbb{EM}, \mathsf{O}, \tau_{\mathsf{o}}, \mathsf{RA}, \tau_{\mathsf{ra}}, \mathsf{ra_t}, \mathsf{ra_s}, \mathsf{AA}, \tau_{\mathsf{aa}}, \mathsf{aa_s}, \mathsf{aa_t})$, *and an object induced sub-model of* EM, *written* $\mathsf{EM}' = (\mathbb{EM}, \mathsf{O}', \tau_{\mathsf{o}}', \mathsf{RA}', \tau_{\mathsf{ra}}', \mathsf{ra_t}', \mathsf{ra_s}', \mathsf{AA}', \tau_{\mathsf{aa}}', \mathsf{aa_s}', \mathsf{aa_t}')$, *suppose that:*

1. EM *conforms to* $\mathbb{EM}$;

2. EM' *satisfies Condition 4;*

3. $\mathbb{EM}$ *satisfies Condition 5;*

then EM' *also conforms to* $\mathbb{EM}$*.*

Proof To check the conformance of EM' to $\mathbb{EM}$, it amounts to checking against the ten conformance conditions in EMF summarized in Section 4.3.

1. Reference typing condition holds in EM' because EM' is an object induced sub-model of EM, i.e., a reference assignment in EM' is also in EM, and EM is conformant to $\mathbb{EM}$, i.e., all reference assignments in EM are type valid.
2. Attribute typing condition holds in EM' because EM' is an object induced sub-model of EM, i.e., an attribute assignment in EM' is also in EM, and EM is conformant to $\mathbb{EM}$, i.e., all attribute assignments in EM are type valid.
3. Reference multiplicity condition holds in EM' because EM' is an object induced sub-model of EM and EM' satisfies Condition 4, i.e., for any object in EM', the set of reference assignments whose source is the object is exactly the same as in EM, and EM is conformant to $\mathbb{EM}$, i.e., the number of reference assignments is in the correct range.
4. Attribute multiplicity condition holds in EM' because EM' is an object induced sub-model of EM, i.e., any object in EM' has exactly the same set of attribute assignments as in EM, and EM is conformant to $\mathbb{EM}$, i.e., the number of attribute assignments is in the correct range.
5. Opposite reference condition: given two references of $\mathbb{EM}$ called $\mathbb{rf}_1$ and $\mathbb{rf}_2$ where $\mathbb{rf}_1.\mathsf{eOpposite} = \mathbb{rf}_2$, two objects $\mathsf{o}_1, \mathsf{o}_2 \in \mathsf{O}'$, and a reference assignment $\mathsf{ra}_1 \in \mathsf{RA}'$ where ${\tau_{\mathsf{ra}}}'(\mathsf{ra}_1) = \mathbb{rf}_1$, ${\mathsf{ra_s}}'(\mathsf{ra}_1) = \mathsf{o}_1$, and ${\mathsf{ra_t}}'(\mathsf{ra}_1) = \mathsf{o}_2$, because EM' is an object induced sub-model of EM, $\mathsf{o}_1, \mathsf{o}_2 \in \mathsf{O}$ and $\mathsf{ra}_1 \in \mathsf{RA}$. Because EM is conformant to $\mathbb{EM}$, there exists a reference assignment $\mathsf{ra}_2 \in \mathsf{RA}$ such that $\tau_{\mathsf{ra}}(\mathsf{ra}_2) = \mathbb{rf}_2$, $\mathsf{ra_s}(\mathsf{ra}_1) = \mathsf{o}_1 = \mathsf{ra_t}(\mathsf{ra}_2)$, and $\mathsf{ra_t}(\mathsf{ra}_1) = \mathsf{o}_2 = \mathsf{ra_s}(\mathsf{ra}_2)$. Again because EM' is an object induced sub-model of EM, we have $\mathsf{ra}_2 \in \mathsf{RA}'$ where ${\tau_{\mathsf{ra}}}'(\mathsf{ra}_2) = \mathbb{rf}_2$, ${\mathsf{ra_t}}'(\mathsf{ra}_2) = \mathsf{o}_1$, and ${\mathsf{ra_s}}'(\mathsf{ra}_2) = \mathsf{o}_2$.
6. Containment condition: on one hand, because EM' is an object induced sub-model of EM, an object o_2 is contained in another object o_1 in EM' if and only if it is also the case in EM; on the other hand, EM is conformant to $\mathbb{EM}$, i.e., no object is contained in more than one object, neither is it contained in itself. Therefore, it is also the case in EM'.
7. Abstract class condition holds in EM' because EM' is an object induced sub-model of EM, i.e., all the objects in EM' are also in EM, and EM is conformant to $\mathbb{EM}$, i.e., none of the objects in EM are instantiated from an abstract class.
8. Uniqueness condition holds in EM' because EM' is an object induced sub-model of EM, i.e., any object in EM' has the subset of reference and attribute assign-

ments as in EM, and EM is conformant to $\mathbb{EM}$, i.e., reference and attribute assignments are never duplicated for unique references and attributes. A subset of a non-multiset is also a non-multiset.

9. Invariant condition: because $\mathbb{EM}$ satisfies Condition 5, all invariants to be checked are forward. Because EM' satisfies Condition 4, for any forward invariant $\mathfrak{einv}$ of $\mathbb{EM}$, an object o in EM whose type is compatible with the context of $\mathfrak{einv}$, the scope of $(\mathfrak{einv}, \mathsf{EM}, \mathsf{o})$ is included in EM' as long as o is. Following Theorem 5, we thus have $\mathsf{eval}(\mathfrak{einv}, \mathsf{EM}, \mathsf{o}) = \mathsf{eval}(\mathfrak{einv}, \mathsf{EM}', \mathsf{o})$. We already know that $\mathfrak{einv}$ holds in EM because it conforms to $\mathbb{EM}$. As a consequence, $\mathfrak{einv}$ also holds in EM'.

□

Lemma 9 *Given an EMF model* $\mathsf{EM} = (\mathbb{EM}, \mathsf{O}, \tau_{\mathsf{o}}, \mathsf{RA}, \tau_{\mathsf{ra}}, \mathsf{ra_t}, \mathsf{ra_s}, \mathsf{AA}, \tau_{\mathsf{aa}}, \mathsf{aa_s}, \mathsf{aa_t})$ *conforming to an EMF metamodel* $\mathbb{EM}$*, and any object induced sub-model of* EM *derived by the EMF model decomposition algorithm, written* $\mathsf{EM}' = (\mathbb{EM}, \mathsf{O}', \tau_{\mathsf{o}}', \mathsf{RA}', \tau_{\mathsf{ra}}', \mathsf{ra_t}', \mathsf{ra_s}', \mathsf{AA}', \tau_{\mathsf{aa}}', \mathsf{aa_s}', \mathsf{aa_t}')$*,* EM' *satisfies Condition 4.*

Proof Let $\mathsf{M} = (\mathbb{M}, \mathsf{N}, \mathsf{A}, \tau, \mathsf{src}, \mathsf{tgt})$ be the corresponding abstract model of EM (i.e., $\mathsf{M} = \mathsf{F}(\mathsf{EM})$) and $\mathsf{M}' = (\mathbb{M}, \mathsf{N}', \mathsf{A}', \tau', \mathsf{src}', \mathsf{tgt}')$ be the sub-model of M, from which EM' is constructed. Assume the setting as illustrated in Figure 10 holds. We demonstrate that EM' satisfies Condition 4 in three steps:

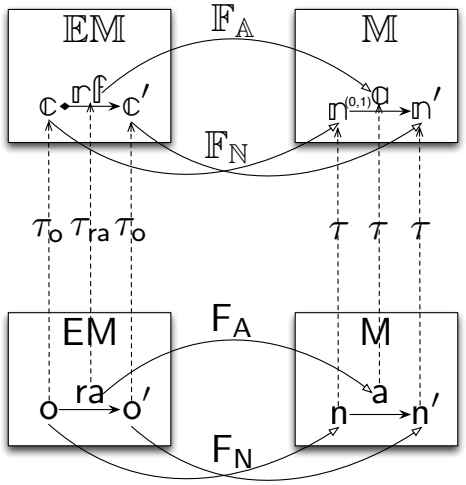

Fig. 10 Setting assumed for the proof of Lemma 9

1. according to step 3 of the EMF model decomposition algorithm, $\mathsf{o} \in \mathsf{O}' \implies \mathsf{n} \in \mathsf{N}'$;
2. according to Theorem 2, model M' satisfies Condition 1, i.e., $\mathsf{n} \in \mathsf{N}' \implies \mathsf{n}' \in \mathsf{N}'$;
3. according to step 3 of the EMF model decomposition algorithm, $\mathsf{n}' \in \mathsf{N}' \implies \mathsf{o}' \in \mathsf{O}'$.

□

A.2 Statistical Table of the Usage of allInstances

Metamodel	Source	# Invariants	# Invariants with allInstances originally	# Invariants with allInstances after normalization	
				monotonic	non-monotonic
UML	OMG	172	0	0	0
MOF	OMG	85	0	0	0
OCL	OMG	46	13	13	0
CORBA	OMG	40	1	1	0
CWM	OMG	90	0	0	0
DiagramDefinition	OMG	16	0	0	0
BLanguage	Academia	9	0	0	0
SAD3	Academia	9	0	0	0
CPFSTool	Academia	27	0	0	0
DeclarativeWorkflow	Academia	23	0	0	0
ER2RE	Academia	59	11	11	0
RBAC	Academia	33	6	0	1
DBLP	Academia	26	7	7	0
SAM	Industry	73	11	7	4
Total	N/A	707	49	39	5

Table 2 Statistics on the Usage of allInstances in Metamodel Specifications