An Analysis of Metamodeling Practices for MOF and OCL

Juan Cadavid, Benoit Combemale, Benoit Baudry

An Analysis of Metamodeling Practices for MOF and OCL

Juan José Cadavid[a], Benoît Combemale[b], Benoit Baudry[b]

[a]*CEA, Saclay, France*
juan.cadavid@cea.fr
[b]*Inria Rennes, France*
{benoit.combemale, benoit.baudry}@inria.fr

Abstract

The definition of a metamodel that precisely captures domain knowledge for effective know-how capitalization is a challenging task. A major obstacle for domain experts who want to build a metamodel is that they must master two radically different languages: an object-oriented, MOF-compliant, modeling language to capture the domain structure and first order logic (the Object Constraint Language) for the definition of well-formedness rules. However, there are no guidelines to assist the conjunct usage of both paradigms, and few tools support it. Consequently, we observe that most metamodels have only an object-oriented domain structure, leading to inaccurate metamodels. In this paper, we perform the first empirical study, which analyzes the current state of practice in metamodels that actually use logical expressions to constrain the structure. We analyze 33 metamodels including 995 rules coming from industry, academia and the Object Management Group, to understand how metamodelers articulate both languages. We implement a set of metrics in the OCLMetrics tool to evaluate the complexity of both parts, as well as the coupling between both. We observe that all metamodels tend to have a small, core subset of concepts, which are constrained by most of the rules, in general the rules are loosely coupled to the structure and we identify the set of OCL constructs actually used in rules.

Keywords: Metamodeling, MOF, OCL.

1. Introduction

Metamodeling is a key activity for capitalizing domain knowledge. A metamodel captures the essential concepts of an engineering domain, providing the basis for the definition of a modeling language. A precise metamodel is essential to drive all the development steps of the modeling language (definition of semantics, construction of editors, etc.) [1]. Yet, the activity of capturing a specific domain expertise in the form of a generic metamodel, is still a craft, where domain experts are the craftsmen. They look at existing practices, interact with stakeholders who build models in that domain and identify the essential concepts to describe abstractions in that domain, providing an initial metamodel. Metamodeling, is thus a labor intensive task, which is not well supported with established best practices and methodologies [2, 3]. Our work aims at observing previous metamodeling experiences, through rigorous empirical inquiry, in order to provide a quantified state of the practice.

Fifteen years ago, the Object Management Group (OMG)[4] introduced the first version of the Meta-Object Facility (MOF) as an attempt to provide a standard metamodeling language, in conjunction with the Object Constraint Language (OCL)[5] to define additional properties through well-formedness rules. Today, in practice MOF is not clearly established as a standard, but a large number of metamodels are defined with two parts: an object-oriented definition of concepts and relationships, and a set of logic-based well-formedness rules. This work's intuition is that the conjunct usage of two languages is cumbersome and thus a major concern for the metamodeling craftsmen (domain experts). Actually, when looking at the most popular metamodel repositories, we find hundreds of metamodels which include only the object-oriented structure, with no well-formedness rules. The consequence is an increased risk of errors in the metamodel [6] and thus errors in all assets that rely on the metamodel. This intuition is thus the hypothesis that guides our scientific method, and the basis for our research questions.

This paper proposes the first extensive empirical analysis of metamodeling practices. The authors published a preliminary proposal at Experiences and Empirical Studies in Software Modeling (EESSMod 2011) held in conjunction with MODELS 2011 [7]. This earlier publication proposed the idea and sketched the workflow to perform the analysis; we present the full survey in this article. The study focuses on the articulation of an object-oriented MOF-compliant language with a logic-based language (OCL) for

the definition of metamodels. We have gathered a collection of 33 metamodels, which combine both paradigms. These metamodels come from diverse backgrounds, in order to effectively cover the state of practice: the OMG (a standardization organism), industry and academia. The object-oriented structures are modeled either with MOF or UML, and all well-formedness rules are modeled with OCL. We analyze the complexity of both parts, as well as different aspects of the coupling relationship. This analysis, based on a set of metamodeling metrics, aims at understanding possible trends in the way metamodeling craftsmen articulate both languages. We observe four phenomena that occur, independent from the metamodel origin:

- Well-formedness rules written in OCL are generally loosely coupled to the underlying object-oriented structure, with a high tendency (87.62% of studied expressions) to define rules referring to 4 or less elements of the domain structure.
- The definition of these rules are not scattered throughout the metamodel, but actually centered in a small subset of classes. In our study, 25 metamodels have a concentration of 80% of their invariants in only one quarter of the metamodel concepts.
- Less than half of the OCL language is used to define invariants. Almost 97% of the studied invariants use a subset of OCL consisting only of 10 concrete expression types out of the 22 specified by the complete language.
- Only 84% of the studied set of invariants (840 out of 995) were written in accordance to the correct syntax of OCL and conforming to the underlying object-oriented structure.

The main contributions of this paper are:

- An empirical inquiry of metamodeling practices, focusing on the combined usage of OCL-based logic formulas and MOF-based object-oriented structures.
- A set of metrics formally defined on MOF and OCL, to quantify the relationship between two paradigms used for metamodeling.
- An openly available set of metamodels using both standards, with a benchmark measuring the aforementioned metrics.

- An Eclipse-based environment to automatically import metamodels and compute metrics on the MOF and OCL parts. OCLMetrics, is the core tool in this environment, which implements our set of metrics for metamodeling.

Target Audience. This article is intended for software engineers who use model-driven techniques and wish to get acquainted with current practices in building metamodels with MOF and OCL, tool vendors interested in providing support for the metamodeling activity with these standards.

Article Structure. In section 2 we illustrate some design issues that arise when metamodeling with MOF and OCL. Then, we give an overview of the two languages. Section 3 formulates our research questions and defines the set of metrics, while section 4 introduces our set of data. Section 5 presents the implementation in the OCLMetrics tool. Section 6 answers our research questions and provides empirical answers. Section 7 presents related work. Finally, section 8 concludes and proposes future directions for this work.

2. Background

This section illustrates the issues that arise from the conjunct usage of two languages for metamodeling, taking Petri nets as illustrative example. Then, we define the most important notions of MOF and OCL, which will support our analysis. It is important to notice that MOF is presented in its specification in two versions, namely Essential MOF (EMOF), which is the core of concepts necessary for the construction of metamodels, and Complete MOF (CMOF), which doesn't add new concepts but merges EMOF with the core definitions of UML. As all the metamodels have been defined with only the essential core of concepts of MOF, in the rest of the paper, mentions to MOF refer actually to Essential MOF.

2.1. Example: Petri nets

The model in figure 1 specifies the concepts and relationships of the Petri net domain structure, expressed in MOF. A PetriNet is composed of several Arcs and several Nodes. Arcs have a source and a target Node, while Nodes can have several incoming and outgoing Arcs. The model distinguishes between two different types of Nodes: Places or Transitions.

The domain structure in figure 1 accurately captures all the concepts that are necessary to build Petri nets, as well as all the valid relationships

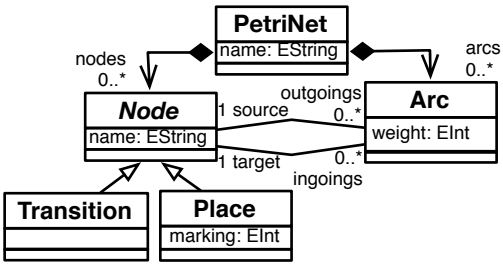


Figure 1: MOF-based domain structure for Petri nets

that can exist between these concepts in a net. However, there can also exist valid instances of this structure that are not valid Petri nets. For example, the model does not prevent the construction of a Petri net in which an arc's source and target are only places (instead of linking a place and a transition). Thus, the sole domain structure of figure 1 is not sufficient to precisely model the specific domain of Petri nets, since it still allows the construction of conforming models that are not valid in this domain.

The domain structure needs to be enhanced with additional properties to capture the domain more precisely. The following well-formedness rules, expressed in OCL, show some mandatory properties of Petri nets.

1. $i1$: Two nodes cannot have the same name.

   ```
   context PetriNet inv : self.nodes->forAll(n1, n2 |
           n1 <> n2 implies n1.name <> n2.name)
   ```

2. $i2$: No arc may connect two places or two transitions.

   ```
   context Arc inv: self.source.oclType() <>
           self.target.oclType()
   ```

3. $i3$: A place's marking must be positive.

   ```
   context Place inv: self.marking >= 0
   ```

4. $i4$: An arc's weight must be strictly positive.

   ```
   context Arc inv: self.weight > 0
   ```

One can notice that $i2$ could have been modeled with MOF by choosing another structure for concepts and relationships. However, the number of concepts and relationships would have increased, hampering the understandability of the metamodel and increasing the distance between the metamodel

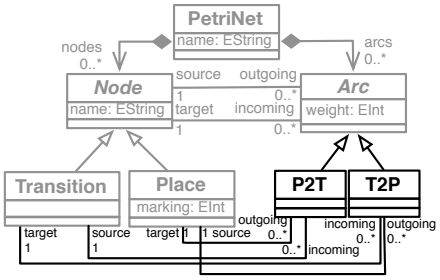


Figure 2: $i2$ expressed in MOF

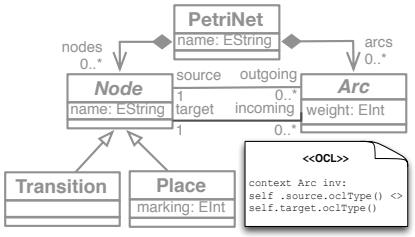


Figure 3: $i2$ expressed in OCL

and a straightforward representation of domain concepts (see figures 2 and 3).

In our study we consider that the metamodel for Petri nets is the composition of the model domain structure and the associated well-formedness rules. We learn from this example that the construction of a precise metamodel, that accurately captures a domain, requires: (i) mastering two formalisms: MOF for concepts and relationships; OCL for additional properties; (ii) building two complimentary views on the domain model; (iii) finding a balance between what is expressed in one or the other formalism, (iv) keeping the views in synchronization, which are expressed in different formalisms. This last point is particularly challenging in case of evolution of one view or the other. One notable case from the OMG and the evolution of the UML standard is that the AssociationEnd class disappeared after version 1.4 in 2003, but as late as in version 2.2, released in 2009, there were still OCL expressions referring to this metaclass [8]. In the same manner, the OCL 2.2 specification depends on MOF 2.0, however we have observed that a particular section of the specification defining the binding between MOF and OCL [5, p.169] makes use of the class ModelElement which only existed until MOF 1.4.

2.2. Definitions

This section defines the terms we use to designate the focus of our analysis of modeling languages based in MOF and OCL. A modeling language captures all the elements that are necessary to build abstract models in a specific business or technical domain. These elements include: a *metamodel* that specifies the concepts and properties that define the structure of models,

a concrete graphical or textual *representation of these concepts*, the *semantics* associated to the concepts and properties, and a set of *generators* for code, documentation, verification, etc.

This paper focuses on the metamodel part of modeling languages. The relationship between a model and a metamodel can be described as shown in figure 4 [9]. Here the conformsTo relation is a predicate function that returns true if all objects in the model are instances of the concepts defined in the metamodel, all relations between objects are valid with respect to relationships defined in the metamodel and if all properties are satisfied.

Definition 1. ***Metamodel****. A metamodel is defined as the composition of:*

- ***Concepts****. The core concepts and attributes that define the domain.*
- ***Relationships****. Relationships that specify how the concepts can be bound together in a model.*
- ***Well-formedness rules****. Additional properties that restrict the way concepts can be assembled to form a valid model.*

In this study, we consider metamodels defined with techniques aligned to the OMG standards, MOF and OCL. We distinguish two parts as defined below.

Definition 2. ***Metamodel under study****. For this work, a metamodel is defined as the composition of:*

- ***Domain structure****. A MOF-compliant model portraying the domain concepts as metaclasses and relationships between them.*
- ***Invariants****. Well-formedness rules that impose invariants over the domain structure and that are expressed in OCL.*

2.3. Summary of MOF and OCL

MOF and OCL are modeling formalisms standardized by the OMG. Ever since its introduction in 1997, MOF has been the metamodeling formalism used by all OMG specifications and it is historically linked to the UML specification, since both standards share a common Core package of constructs. OCL emerged as a component of the UML 1.4 specification and later became

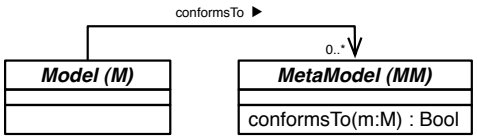


Figure 4: Model & MetaModel Definition with Class Diagram Notation

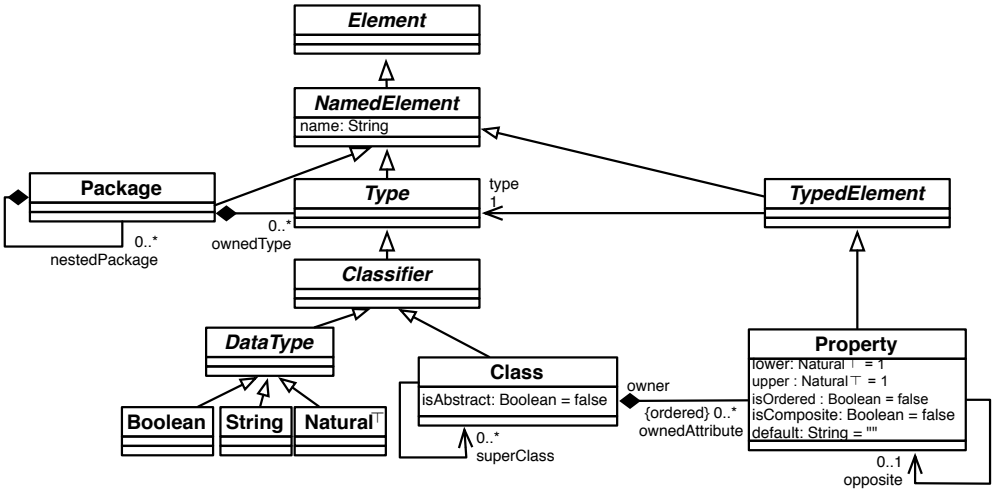


Figure 5: The MOF 2.0 Core with Class Diagram Notation

an independent standard with application domains outside UML. Today it is the formalism employed for diverse activities such as model transformations, automatic generation of instances and others within the Model-Driven Engineering realm [10, 11, 12]. In this section, we discuss the main MOF and OCL concepts that are necessary to perform our analysis, as well as the connection between the two formalisms.

Figure 5 displays a subset of (or an abstraction of) the structure of MOF version 2.0 [4]. MOF allows to specify the concepts of a metamodel in a Package. This Package contains Classes and Properties to model the concepts and relationships. The Properties of a Class are typed by a Classifier, which can be either a DataType Boolean, String or Natural; or another Class.

Figure 6 displays a subset of the structure of OCL expressions [5] that can be used to constrain the structure defined with MOF. The most noticeable constructs for OCL expressions are: the ability to declare Variables, whose

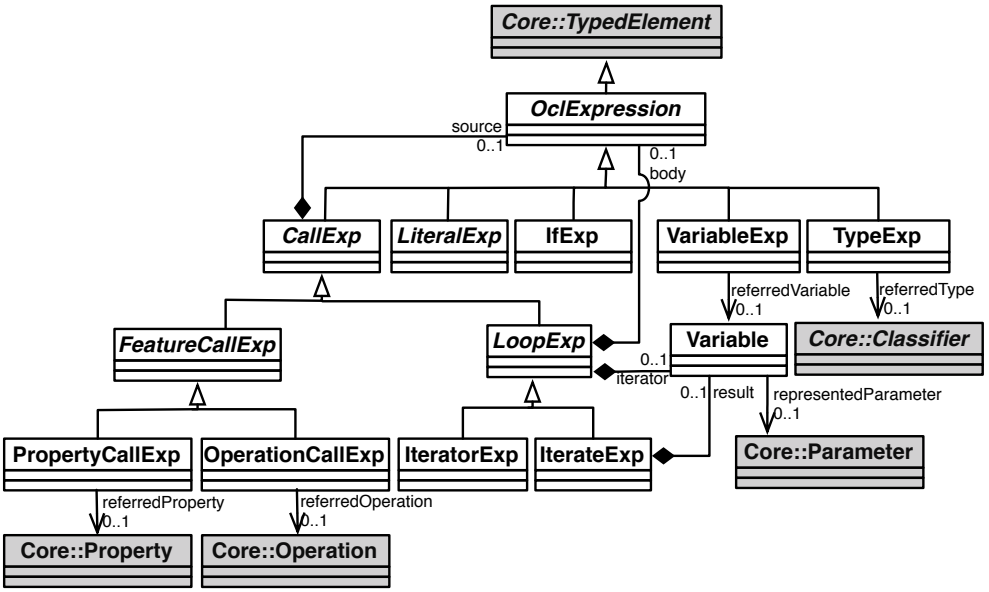


Figure 6: OCL Expression metamodel

type is a concept modeled with MOF; the ability to use control structures such as IfExp and LoopExp; the ability to have composite OCL expressions, through CallExps.

Figure 7 illustrates how OCL and MOF formalisms are bound to each other [5, p.169]. This figure specifies that it is possible to define Constraints on MOF Elements (everything in MOF is an Element, cf. figure 5). One particular subtype of Element is important for metamodeling: Classifier, as it appears previously in figure 5. Constraints can be defined as Expressions, and one particular type of expression is ExpressionInOCL, an expression whose the body is defined with OCL. The existence of this binding between formalisms is essential for metamodeling: this is how two different formalisms can be smoothly integrated in the construction of a metamodel. This binding is also what allows us to automatically analyze metamodels built with MOF and OCL. Notice that ModelElement class in figure 7 refers to the Element metaclass in figure 5.

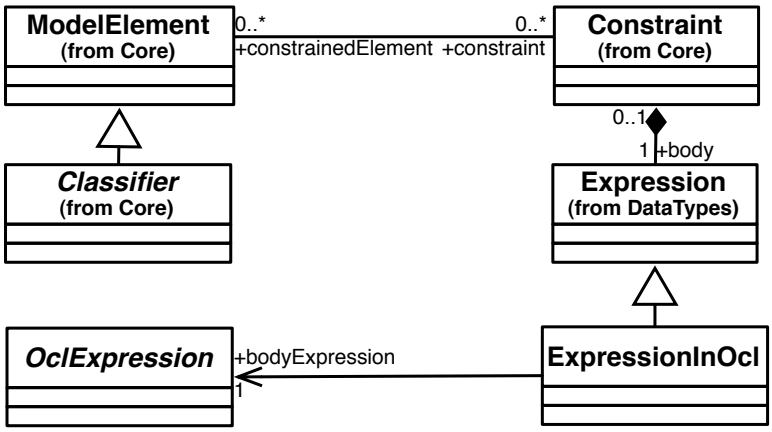


Figure 7: OCL and MOF binding

3. Research Questions and Metrics over MOF and OCL in metamodels

In this section we introduce the research questions that we address to understand the usage of MOF and OCL for metamodeling. We also discuss the measurable attributes we compute defining reusable metrics and providing answers to the research questions. All metrics are defined on the basis of data sets that we gather from the MOF domain structure and the associated OCL invariants. We use OCL itself as the formalism to define data sets and metrics, as it provides an interesting trade-off between understandability and formality, and it has been successfully used before in metrics both at code and model levels [13].

3.1. Research questions

3.1.1. Q1: How consistent are OCL expressions with respect to the OCL language syntax and the domain structure?

As explained in section 2, the domain structure part, concretely the modeling of concepts and relationships, takes high priority in the development of modeling languages, resulting in a high use of MOF compared to the use of OCL. This initial question aims at providing an overview of the validity of invariants defined over MOF structures and a global comparison of OCL usage among all metamodels. We check to what extent expressions in OCL

defined in the sample metamodels are syntactically correct with respect to the MOF concepts and relations definitions. For all cases of invalid expressions, we classify and list the causes of errors, providing an initial qualitative assessment of the collect metamodels.

3.1.2. Q2. How balanced is the distribution of OCL invariants definitions in the domain structure?

As shown in the MOF-OCL binding of figure 7, every invariant is defined over a concept class in the metamodel, which forms the context of the rule. Since the domain structure of a metamodel defines the set of classes in the metamodel, we wonder whether all concept classes in the domain structure equally serve as context, *i.e.* if well-formedness rules are equally scattered in the metamodel or if they tend to concentrate on a subset of the structure. To answer this question, we will develop a metric to measure the proportion of the number of invariants defined for every context of the total set of invariants of the metamodel.

3.1.3. Q3. How pervasive are OCL invariants in the domain structure?

OCL invariants express relationships between concepts and properties that are captured in the domain structure. This question aims at understanding the level of coupling between invariants and structure, and how this coupling varies among the different structures. Some metamodels contain lengthy and complex invariants, while others seem to define them using simple expressions. We will compute the *expansion* in the domain structure of each invariant, *i.e.* the classes and attributes that the invariant captures. We define a metric to measure the size of this expansion, thus quantifying the complexity of OCL invariants.

3.1.4. Q4. What is the extent of the usage of the OCL language constructs in metamodels?

As shown in subsection 2.3, OCL is a rich language providing several types of expressions, albeit the critique raised by its usability and ambiguity shortcomings [14, 15]. Consequently, many OCL engines implement the standard with a certain degree of inaccuracy, as demonstrated by Gogolla et al. [11]. Our intuition is that OCL contains some essential concepts for metamodeling, while some other concepts are rarely used. This question aims at identifying whether there is such a core subset of OCL. We will compute

the *expansion* in the OCL language for each invariant, *i.e.* identify the constructs of the language used in every expression. We will quantify the usage for every invariant by looking at the number of employed constructs.

3.2. MOF and OCL data sets

This subsection defines formally the data sets for MOF models, OCL invariants and the binding produced when both are used together. This will allow us to measure the attributes discussed in the research questions.

MOF data sets. All data for a metamodel are gathered in the context of a package (figure 5). Our metrics manipulate the set of properties in a class (classContent()), the set of classes in one package (packageContent()), the set of all classes in the metamodel (packageAllContent() recursively collects all the classes from contained subpackages). All these MOF data sets are formally defined as follows:

```
context Package def :
packageContent() : Collection <Class> =
    self.ownedType->select(t | t.oclIsTypeOf(Class))
```

Listing 1: 'Definition of the packageContent dataset'

```
context Package def :
packageAllContent() : Collection <Class> =
    self.ownedType->select(t |
    t.oclIsTypeOf(Class)).union(self
    .nestedPakages->collect(p |
p.packageAllContent()))
```

Listing 2: 'Definition of the packageAllContent dataset'

```
context Class def :
classContent() : Collection <Property> =
    self.ownedAttribute
```

Listing 3: 'Definition of the classContent dataset'

OCL data sets. Metrics about OCL expressions are defined using two data sets gathered from instances of the OCL expression metamodel (figure 6):

- computeOCLE(): the set of all OclExpression instances that are manipulated in an ExpressionInOCL.

- expansionInDS(): the set of different properties of the referenced domain structure that are used in an OCL expression. Note that the context for computing this set is ExpressionInOCL, and we assume that this expression is an invariant, since OCL expressions considered in this study are only invariants (not pre- or post-conditions).

- expansionInOCL(): the set of the OCL constructs used by the OclExpressions returned by computeOCLE().

These OCL data sets are formally defined as follows:

```
%computeOCLE is a helper function that returns all
    OclExpression instances that are manipulated in an
    ExpressionInOCL.
computeOCLE() : Collection <OclExpression>
```

Listing 4: 'Definition of the computeOCLE data set.'

```
context ExpressionInOCL def :
expansionInDS() : Set <Property> =
  self.computeOCLE() -> select(pce |
     pce.oclIsTypeOf(PropertyCallExpr))
      -> collect (pce | pce.asOclType(PropertyCallExpr)
      .referredProperty)->asSet()
```

Listing 5: 'Definition of the expansionInDS data set.'

```
context ExpressionInOCL def :
expansionInOCL() : Set <OclExpression> =
  self.computeOCLE() -> iterate(expr , acc :
     Collection<OclExpression> = Set{} |
        (not acc->exists(e | e.oclType() =
           expr.oclType())) implies acc.add(expr))
```

Listing 6: 'Definition of the expansionInOCL data set'

MOF-OCL binding data sets. Metrics about the binding between the MOF structure and the OCL expressions:

- classInvariants(): Retrieves the invariants of a specific class of the metamodel.

```
context Class::classInvariants() : Set<ExpressionInOcl>
def:
self.constraint->select(c |
    c.body.oclIsTypeOf(ExpressionInOcl))
->collect(body.asOclType(ExpressionInOcl))
```

Listing 7: 'Definition of the classInvariants data set.'

We have developed a tool that automatically analyzes a metamodel to gather the MOF, OCL and MOF-OCL data sets, which will be detailed in section 5.

3.3. Metric Definitions

Based on the defined data sets, we can compute the following metrics to answer our research questions.

Definition 3. ***Size of Domain Structure (SDS)****. The size of a domain structure is the sum of the number of classes and the number of properties (i.e. attributes and association ends) in the metamodel. This is formally defined as follows:*

```
context Package::SDS : Integer derive :
self.packageAllContent()->iterate(acc : Integer = 0, c |
    acc +
c.classContent()->size()) +
    self.packageAllContent()->size()
```

Listing 8: Definition of the Size of a Domain Structure (SDS)

Definition 4. ***Size of Specified Invariant Set (SSIS)****. The number of invariants defined in OCL defined over the classes and its properties (i.e. attributes and association ends) of the metamodel. This is formally defined as follows:*

```
context Package::SSIS : Integer derive :
self.packageContents()->iterate(acc : Integer = 0, i | acc
    + i.classInvariants()->size())
```

Listing 9: Size of Specified Invariant Set (SSIS)

Definition 5. ***Size of Parsed Invariant Set (SPIS)****. We count the number of elements in the subset of the specified invariants set that can be successfully parsed according to our reference OCL parser embedded in our metrics computation tool. This will be further detailed in section 5.*

```
context Package::SPIS : Integer derive :
self.packageContents()->iterate(acc : Integer = 0, i | acc
   + i.classInvariants()->select(e : ExpressionInOcl | not
   e.oclIsTypeOf(BooleanLiteral)
and not e.oclIsUndefined())->size())
```

Listing 10: Size of Parsed Invariant Set (SPIS)

Definition 6. ***Number of Invariants Defined by Context (NIC)****. This is the equivalent of the precedent metric at the class level. It is simply the count of invariants defined with a class as context.*

```
context Class::NIC : Integer derive :
self.classInvariants()->size()
```

Listing 11: Number of Invariants Defined by Context (NIC)

Definition 7. ***Invariant Complexity w.r.t. a Domain Structure (IC_DS)****. The complexity of an invariant i with respect to a domain structure IC_DS is the number of different roles defined in the domain structure that are used in i. IC is thus computed as the size of* expansionInDS()*.*

```
context ExpressionInOCL::IC_DS : Integer
derive : self.expansionInDS()->size()
```

Listing 12: Definition of the Invariant Complexity w.r.t. a Domain Structure (IC_DS)

Definition 8. ***Context Complexity w.r.t. a Domain Structure (CC_DS)****. This metric computes the number of different elements from the domain structure that are used in all the invariants of one class.*

```
context Class::CC_DS : Integer
derive :
(self.classInvariants->iterate(acc : Collection<Property>
   = Set{}, i | acc.union(i.expansionInDS()))
        ->asSet()->size()
```

Listing 13: Definition of the Class Complexity w.r.t. a Domain Structure (CC_DS)

Definition 9. ***Invariant Complexity w.r.t. OCL (IC_OCL).*** *The complexity of an invariant i with respect to the OCL language is the number of unique OCL constructs used in the invariant.*

```
context ExpressionInOCL::IC_OCL : Integer
derive : self.expansionInOCL() -> size()
```

Listing 14: Definition of the Invariant Complexity w.r.t. OCL (IC_OCL)

Definition 10. ***Context Complexity w.r.t. OCL (CC_OCL).*** *This is the number of unique OCL constructs that are used by all invariants of a class.*

```
context Class::CC_OCL : Integer
derive :
(self.classInvariants->iterate(acc :
   Collection<OclExpression> = Set{}, i |
   acc.union(i.expansionInOCL()))
        ->asSet()->size()
```

Listing 15: Definition of the Class Complexity w.r.t. OCL (CC_OCL)

3.4. Examples

This section illustrates the computation of some of the metrics with the Petri nets example.

Example 1. *Let ds be the Petri nets domain structure presented in section 2. The value of SDS is calculated as follows.*

$$\begin{aligned} SDS(ds) \;=\; & Set\{\text{PetriNet}, \text{Node}, \text{Arc}, \text{Transition}, \text{Place}\}->size()+ \\ & Set\{\text{name}, \text{name}, \text{weight}, \text{marking}, \text{source}, \text{target}, \\ & \text{ingoings}, \text{outgoings}\}->size() = 13 \end{aligned}$$

Example 2. *Consider the invariants of Petri nets $i1$, $i2$, $i3$, $i4$. The value of IC_DS is calculated as follows.*

$$\begin{array}{rcl} IC\_DS(i1) & = & IC\_DS(i2) = 2 \\ IC\_DS(i3) & = & IC\_DS(i4) = 1 \end{array}$$

Invariants $i1$ and $i2$ deal each one with two MOF properties, whereas $i3$ and $i4$ deal with one, therefore its complexity with respect to the domain structure is two and one respectively.

Example 3. *Consider the invariants of Petri nets $i1$, $i2$, $i3$, $i4$. The value of IC_OCL is calculated as follows.*

$$\begin{array}{rcl} IC\_OCL(i1) & = & |\{\text{OperationCallExpr}, \\ & & \text{PropertyCallExpr}, \text{VariableExp}\}| = 3 \\ IC\_OCL(i2) & = & |\{\text{OperationCallExpr}, \text{PropertyCallExpr}\}| = 2 \\ IC\_OCL(i3) & = & |\{\text{OperationCallExpr}, \text{IntegerLiteralExpr}, \\ & & \text{PropertyCallExpr}\}| = 3 \\ IC\_OCL(i4) & = & |\{\text{OperationCallExpr}, \text{IntegerLiteralExpr}, \\ & & \text{PropertyCallExpr}\}| = 3 \end{array}$$

4. Experimental setup

Answering our research questions requires a sample of metamodels from repositories in diverse backgrounds. Accessing such a sample proved from the start to be a challenge. There exist multiple open repositories [1], but these contain exclusively metamodels without any well-formedness rules. There are very few metamodels making use of MOF and OCL for metamodeling. Data collection was thus an important step for our analysis.

Our sample data comes from standard bodies, academia and industry altogether. We collected standard metamodels from the OMG[2]. For academic and industry metamodels, we asked the model-driven engineering community if it could provide data. We made a call for participation on the PlanetMDE mailing list dedicated to the dissemination of news and information about model-driven engineering, counting over 700 subscribers in July 2012. We

[1] The most extensive repository known to us is the AtlanMod Metamodel Repository, containing 305 metamodels.

[2] http://www.omg.org

received replies from Europe and North America, on the basis of which we constituted our academic and industry data sets.

We filtered our initial set of metamodels in order to keep the ones that could be automatically processed for analysis. As a first criterion for selection, we only considered metamodels that make use of OCL to define invariants. Additionally, we considered only modeling language specifications containing metamodels defined with formalisms aligned with the MOF standard. In some cases, we have got metamodels expressed in UML, which in turn conform themselves to MOF. Metamodels created with more complex mechanisms, such as UML Profiles, were not taken into account. Table 1 shows the studied specifications, each one containing one or more metamodels. We have divided our data samples in three groups according to their origin.

Standards community: The first group comes from the OMG. The OMG defines standards across several domains, such as object-oriented development, real-time systems and embedded systems, extending the boundaries of modeling to specific domains such as finance and healthcare. An OMG specification is a public, textual document proposed by the OMG to define one or more metamodels. The analyzed specifications are:

- **UML (Unified Modeling Language) version 2.2** [16]. It uses a structure of 13 packages to define different types of diagrams to represent the different views of a system, as well as the extension mechanism through profiles. Each one of these packages is regarded in this study as an independent metamodel. We consider the Ecore[3] metamodels provided by the Eclipse UML2 project, version 3.0.1 [18]. To our knowledge, this project constitutes the best analyzable form of the UML specification openly available; it has been constructed according to the UML 2.2 specification.

- **MOF (Meta-Object Facility) version 2.0** [4]. The specification that created the standard for the exchange of metadata, therefore creating the language for metamodels themselves. It was created from the modeling foundations of UML and comprises two metamodels, Essential MOF (EMOF) and Complete MOF (CMOF).

[3]Ecore is an implementation of MOF provided by the *Eclipse Modeling Framework* [17]

- **OCL version 2.2** [5]. We analyze the specification of the OCL language itself, which contains four metamodels. An Ecore implementation of the four metamodels is available from the Eclipse OCL project [19]. We consider only the two metamodels that contain OCL invariants, namely OCL types and OCL expressions.

- **CORBA Component Model version 1.0** [20]. An Ecore implementation of the four metamodels in this specification is provided by the SourceForge CORBA project [21]. We introduced few minor modifications to align this metamodel with the one defined in the standard specification. OCL invariants are defined only for three metamodels, in which we focus for our analysis.

- **Diagram Definition (DD) version 1.1** [22]. Standard providing a basis for creating and interchanging graphical notations. It contains two metamodels: diagram common elements and diagram graphics.

- **Common Warehouse Metamodel (CWM) version 1.1** [23]. Specification to enable interchange of warehouse and business intelligence metadata between warehouse tools. It contains one metamodel, structured in 19 packages.

Academic research community: The following group presents metamodels taken from research in academic groups and projects.

- **B language** metamodel created at IMAG.

- **SAD3** is a software architecture component model created at ENSTA Bretagne.

- **CPFSTool** is a metamodel and tool developed for the specification of patterns for security requirements engineering at the University Duisburg-Essen.

- **Declarative Workflow** is a metamodel describing an approach to define workflows in a declarative way. It has been developed with USE at University of Rostock and University of Bremen.

- **ER 2 RE** is a metamodel describing a model transformation from an entity-relationship scheme to a relational model. It has been developed with USE at University of Bremen.

- **RBAC** is a metamodel describing the Role-Based Access Control security standard. It has been developed with USE at University of Bremen.
- **HRC (Heterogeneous Rich Components)** is a metamodel created within the european research project SPEEDS with the Kermeta metamodeling environment [24].

Industrial community: This group contains metamodels developed in enterprises using model-driven techniques for their software projects.

- **MTEP** and **XMS** are metamodels created by Thomson Video Networks for encoding standards for video hardware.
- **SAM** is a metamodel from the Topcased open source software project.

Table 1 details a list of standard specifications of modeling languages coming from different sources. Each specification contains one or more metamodels. The first two columns contain the name and group; The third column counts the number of metamodels. In the OMG group, specifications define large modeling languages, normally structured in packages, therefore we treat each one of these as a separate metamodel. In the remaining cases, each specification contains only one metamodel. The fourth column mentions the formalism used to express invariants. As expected, we chose specifications using OCL. The fifth column shows the different standards that exist to specify the domain structure. The sixth column presents the format for expressing invariants in OCL. These are found either as separate .ocl text files or embedded in .ecore as annotations. We present in table 2 each one of the metamodels analyzed, assigning an ID that will be later used in the layout of our results.

We make available this set of metamodels based on MOF and OCL, as one of the contributions of our work, should the community wish to carry further studies involving metamodels expressed in both standards. They are available for download in the web page created for our study[4].

5. Automatic analysis of MOF and OCL in metamodels

We have developed a tool to automatically compute metrics on both parts of a metamodel and provide data for our empirical enquiry. All the metamod-

[4]http://www.irisa.fr/triskell/Software/protos/mof-ocl-study/sourceMetamodels/

Table 1: Specifications containing sample metamodels.

Name	Source	Meta-models	Expression of Constraints	Domain Structure format	OCL invariants format
UML	OMG	13	Natural Language and OCL	Ecore	Annotations in Ecore
CCM	OMG	4	Natural Language and OCL	Ecore	Text in documentation
OCL	OMG	4	Natural Language and OCL	Ecore	Text in documentation
MOF	OMG	2	Natural Language and OCL	CMOF	.ocl text file
CWM	OMG	1	Natural Language and OCL	Ecore	Tex-Mext in documentation
DD	OMG	3	Natural Language and OCL	CMOF	Annotations in Ecore
B language	Academic Research	1	OCL	Ecore	.ocl text file
SAD3	Academic Research	1	OCL	Ecore	.ocl text file
CPFSTool	Academic Research	1	OCL	Textual UML	.ocl text file
Declarative workflow	Academic Research	1	OCL	USE	embedded in USE file
ER2RE	Academic Research	1	OCL	USE	embedded in USE file
RBAC	Academic Research	1	OCL	USE	embedded in USE file
HRC	Academic Research	1	OCL and Kermeta	Ecore and Kermeta	.ocl text file
MTEP	Industry	1	OCL	Ecore	.ocl text file
XMS	Industry	1	OCL	Ecore	.ocl text file
SAM	Industry	1	OCL	Ecore	.ocl text file

Table 2: The analyzed Domain-Specific Modeling Languages.

Group	ID	Name	Number of Metaclasses
Standards	uml1	UML Classes	113
	uml2	UML Profiles	60
	uml3	UML Common Behaviors	186
	uml4	UML Activities	143
	uml5	UML Information Flows	84
	uml6	UML Composite Structures	75
	uml7	UML Interactions	115
	uml8	UML Deployments	64
	uml9	UML State Machines	95
	uml10	UML Components	20
	uml11	UML Templates	30
	uml12	UML Actions	113
	uml13	UML Use Cases	17
	cor1	CORBA Component IDL	14
	cor2	CORBA Base IDL	30
	cor3	CORBA Deployment	17
	ocl1	OCL Types	12
	ocl2	OCL Expressions	25
	emof	Essential MOF	74
	cmof	Complete MOF	74
	dc	Diagram Definition Common	4
	dg	Diagram Definition Graphics	35
	cwm	Common Warehouse Metamodel	251
Academy	b	B language	34
	sad3	SAD	41
	cspf	CSPFTool	18
	dwf	Declarative Workflow	39
	erre	ER to RE transformation	18
	rbac	RBAC	11
	hrc	Heterogeneous Rich Components	135
Industry	mtep	Thomson MTEP	18
	xms	Thomson XMS	55
	sam	Topcased SAM	48

els we gathered had different formats. Thus, our measurement environment has a preprocessing step that transforms all these formats into a common one over which we compute metrics. The architecture of the tool is extensible through the definition of plug-ins to allow future experiments with metamodel formats that are not currently supported. This section presents the data flow for analysis as well as the global architecture of the tool.

5.1. The Global Process for Analysis Automation

Figure 8 shows the overall process to analyze a metamodel. The process is composed of three activities with their own tools:

1. If the OCL invariants are not defined as a model conforming to the OCL metamodel (extension .oclxmi in figure 8), the first activity consists of parsing the invariants to build a model (activity OCL Parsing in figure 8) linked to the domain structure of the metamodel, which is given in the Ecore format, which is a lightweight implementation of MOF [17], providing equally an XMI-based persistence mechanism. Parsing must be defined depending on the input format of the OCL invariants (sixth column of table 1).
2. The next step consists of using OCLMetrics, the tool we have developed to automatically compute the metrics over the metamodel (activity Metrics Computation in figure 8). OCLMetrics takes as an input the metamodel composed of the domain structure expressed in Ecore, and the invariants expressed in OCL. Then, OCLMetrics produces a CSV file containing all the metric values for the input metamodel.
3. The metric values are finally analyzed with R[5] (activity Statistical Analysis in figure 8). R is an open-source language for statistical applications, which provides several functionalities to run analysis and create plots, both one-variable and multi-variable. For metamodel analysis, we provide a set of generic scripts that could be used for any CSV file produced with OCLMetrics. These scripts automate the production of graphics (statistical charts in figure 8 in terms of *bar plots*, *boxplots* and *treemaps*) to help the metrics analysis over the metamodel.

[5]R, cf. http://www.r-project.org/

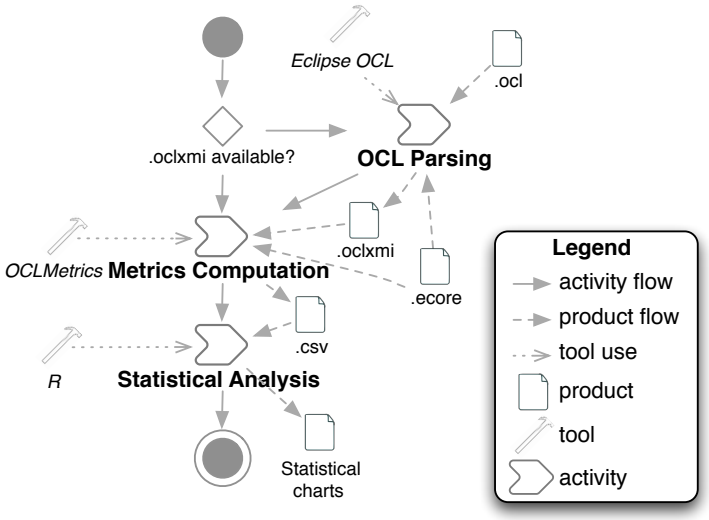


Figure 8: SPEM Process for Metamodel Automatic Analysis

5.2. Metamodeling Analysis Environment

Our metamodeling analysis environment has been designed with an extensible architecture. It has been developed as a set of plug-ins for Eclipse. Figure 9 shows an extract of the architecture. The components of this architecture are the following:

5.2.1. Preprocessing Core

This component realizes the preprocessing step specified in the previous subsection. It provides utilities for preprocessing the different possible formats:

- An extractor of OCL constraints from Ecore metamodels, using Xpath.
- An extractor of OCL constraints from XMI files based on the UML schema, using Xpath.
- An extractor of OCL constraints from XMI files based on the CMOF schema, using Xpath.
- A transformer from MOF 1.1 to Ecore.
- A transformer from CMOF (Complete MOF) 2.0 to Ecore.

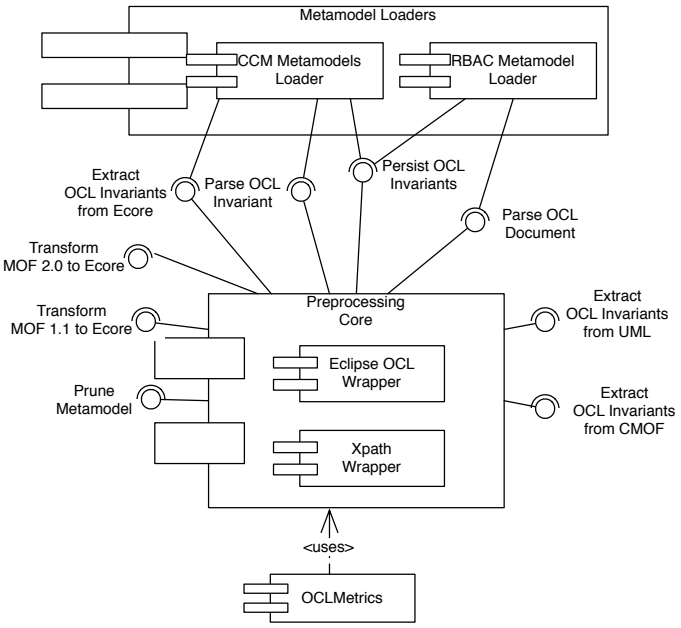


Figure 9: Extract of the architecture of the metamodeling analysis environment.

- A wrapper for the Eclipse OCL parser of individual constraints
- A wrapper for the Eclipse OCL parser of documents of constraints (.ocl files)
- A wrapper of the Eclipse OCL persisting mechanism in order to save a parsed version of an OCL expression as an instance of the OCL abstract syntax metamodel (.oclxmi files).
- A wrapper to the Kermeta Metamodel Pruner. This is an utility that allows the preprocessor to prune a metamodel to include only a set of required classes and properties given as input [25].

5.2.2. Metamodel Loaders

For our study, we have created loaders for each one of the specifications in our experimental setup; for example, the UML Metamodels Loader uses the preprocessing core's utilities to load the 13 UML metamodels. Each loader is an Eclipse plug-in that makes use of the extension point provided by the preprocessing core component. Every metamodel can define its MOF and OCL artifacts in a single file, a file for each, or multiple files for both. Each

Table 3: Invoked services by metamodel loader.

Metamodel loader	Invoked services
UML	Prune metamodel, extract OCL invariants from UML, persist OCL invariants
CCM	Extract OCL invariants from Ecore, parse OCL invariant, persist OCL invariants
CWM	Transform XMI 1.1 to Ecore, parse OCL Document, persist OCL invariants
OCL	Extract OCL invariants from Ecore, parse OCL invariant, persist OCL invariants
MOF	Prune metamodel, parse OCL document, persist OCL invariant
DD	Extract OCL invariants from CMOF, parse OCL invariant, persist OCL invariants
B	Extract OCL invariants from Ecore, parse OCL invariant, parse OCL document, persist OCL invariants
OCL Document loader	Parse OCL document, persist OCL invariants

metamodel loader takes care of loading this set of files, and then it invokes the utilities in order to extract constraints, validate them and parse them, and then persist them in order to have them in the .oclxmi format, which is suitable to perform the metrics analysis.

In the case of specifications DWF, CSPF, ERRE, HRC, RBAC, MTEP, XMS, SAM, which provide the domain structure in a single .ecore file and define the integral set of invariants a single .ocl file, a single loader was built (called "OCL Document Loader"). Loaders for additional metamodels can be built to include them in the sample and run OCLMetrics on them. It only takes declaring the extension to the preprocessor core's extension point in the plug-in's configuration file.

In the figure, as an example the Corba Component Metamodel uses the service to extract the OCL invariants embedded in an Ecore file, as well as the parsing of each individual invariant and persistence. The RBAC Metamodel, as there was an available source file of OCL invariants as an .ocl file, uses the service of parsing such file type and secondly to persist them. Table 3 shows the used components by each one of the metamodel loaders.

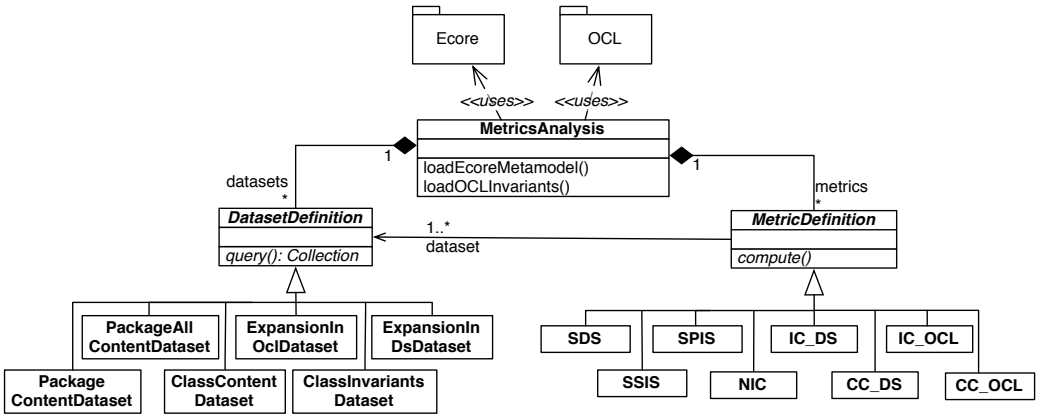


Figure 10: UML Class diagram for OCLMetrics

5.2.3. OCLMetrics

At the root of the architecture, the OCLMetrics component makes use of the Preprocessing Core, because it depends on the output format of the metamodels (.ecore for the MOF part and .oclxmi for the OCL part) to perform the measurement of metrics. The data sets and metrics specified in the previous section were implemented in the OCLMetrics tool. OCLMetrics considers a domain structure defined in Ecore and the associated OCL. The tool can then analyze both parts of a metamodel, to automatically extract the data sets and compute the metrics.

Figure 10 shows the class diagram of the OCLMetrics tool. For every analyzed metamodel, the main class MetricsAnalysis loads the domain structure (method loadEcoreMetamodel()) and the associated OCL invariants (method loadOCLInvariants()). MetricsAnalysis relies on the binding between MOF/Ecore and OCL as shown in figure 7 to load and manipulate the corresponding models. The tool uses the Ecore and OCL metamodels defined in the packages Ecore and OCL. The class MetricsAnalysis mainly defines data sets (reference datasets) and metrics (reference metrics) specified in the previous section. Each data set (class DatasetDefinition) is implemented in the method query() that computes the resulting collection for a given metamodel. A data set is used by a metric (class MetricDefinition) whose definition is implemented in the method compute().

This component also creates the following .csv files of data:

- TabAllInvariants: Table measuring the metrics at the invariant level;

each record corresponds to an invariant, described by an identifier given by the metamodel where it is defined, followed by the context class and then an unique ID number. This is followed by all the metrics at the invariant level.

- TabAllClasses: Table measuring the metrics at the class level; each record corresponds to a class in a metamodel, described by an identifier given by the metamodel where it is defined and then an unique ID number. This is followed by all the metrics at the class level, which summarize all the invariants that have been defined with this class as context.

- TabAllMetamodels: Table measuring the metrics at the metamodel level; each record corresponds to a metamodel, describing all the metrics at the metamodel level, which summarize all the invariants that have been defined within this metamodel.

As a whole, OCLMetrics consists of 11 classes, with a total of 486 lines of Java code. It can be extended with new data sets and metric definitions by implementing new subclasses to the classes DatasetDefinition and MetricDefinition respectively. The entire metamodeling analysis environment is available for download in the web page created for our study[6].

6. Experimental analysis

In this section we compute the metrics defined in section 3 in order to answer our research questions, and discuss potential threats to validity.

6.1. Results

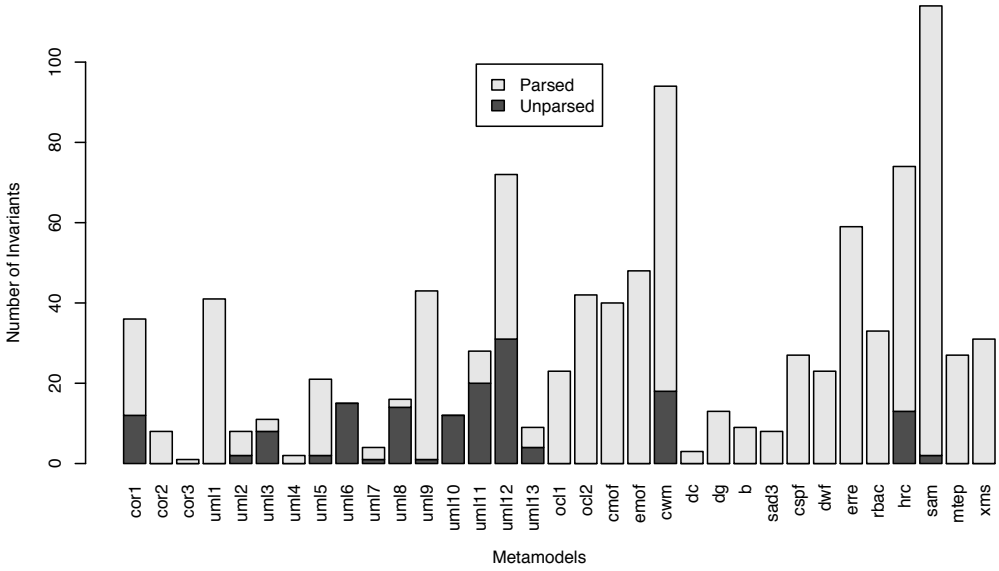
For each question, we display some metrics relevant to the answer and comment the results.

6.1.1. Q1: How consistent are OCL expressions with respect to the OCL language syntax and the domain structure?

To answer this question we run a preprocessing step of our analysis process on each metamodel. We learn that OCL invariants are not always syntactically correct, as a number of invariants do not pass the syntactic or semantic

[6]http://www.irisa.fr/triskell/Software/protos/mof-ocl-study/OCLMetrics/

Figure 11: Successfully parsed vs. unable to parse invariants



validation of the parser. Figure 11 shows this phenomenon, comparing the size of the specified invariants set (SSIS metric) versus the size of the parsed invariants set (SPIS) that could not be parsed, for each metamodel.

When analyzing the 995 invariants of 33 metamodels, 567 were successfully parsed. Regarding the 428 invariants that could not be successfully parsed at first, in the case of 273 invariants we have been able to identify the source of the problem and we have fixed these invariants. This leaves us with a total of 840 parsed invariants. Throughout this process we have observed the following issues.

Different storage formats. Our data setup includes metamodels in different storage formats. Although they are all aligned with MOF, as seen in table 2, different formats exist to express the domain structure, and we also realize there is no single standard format to store OCL expressions for a metamodel. Besides OCL text files, invariants are also added as annotations; however these only consist of maps of string-to-string entries, which can themselves present different schemas. Our preprocessing program automatically detects the format and proceeds to parse and produce the previously mentioned output.

Table 4: Corrected errors in OCL invariants.

Corrected errors	**Number of Occurrences**
Missing parenthesis	209
Notation for enumeration literals	92
Missing mandatory typecast (oclAsType())	45
Typos in pointers to metaclasses and properties	42
Missing variable in forAll body	30
Typos in OCL operations invocation	28
Use of '->' instead of '.' for non-collection properties	15
Use of unescaped OCL keywords	13
Use of '.' as a shortcut for 'collect'	9
Undeclared type of variable	9
'if' expression without 'else' and 'endif'	5
Use of 'notEmpty' and 'isEmpty' for non-collection properties instead of oclIsUndefined()	4
Treating of boolean values as literals '#true' and '#false'	3
Use of 'union' instead of 'concat' to concatenate strings	2

Different OCL syntaxes. Different parsers allow or reject certain OCL constructs [11]. To enable automation analysis of the OCL expressions, such variations must be streamlined to satisfy the precise syntax required for Eclipse OCL; this was performed by replacing the unrecognized constructs by its accepted equivalents; for example, the use of the minus "-" operator to exclude elements from a collection, instead of the *exclude* operation.

Errors in invariants. In many cases, OCL invariants are added to a metamodel with the sole purpose of documentation and might not be checked syntactic validity. The studied sets of invariants from the selected specifications contained incorrect OCL expressions, containing errors from syntax (invalid use of OCL constructs) or semantics (references to non-existent model elements from the domain structure). Table 5 presents simple errors, which we could fix, as well as those that could not be fixed, since it would require knowledge from the domain expert.

Table 5: Unfixable errors in OCL invariants.

Errors remaining incorrect	**Number of Occurrences**
Pointers to nonexistent properties/operations	161
Invariants with a context metaclass in an outside metamodel	3
Reference to undefined stereotypes	2

Most of the 155 invariants that we could not fix do not parse because of pointers to properties or operations that do not exist in the domain structure. In some cases these properties were defined in previous versions of the domain structure and we do not know how or if they have been replaced in the version under study. One notable example can be found in the OCL Expressions modeling language (o2): it defines 14 invariants invoking the conformsTo operation, which does not exist in the OCL Expressions domain structure, but rather in a foreign imported package which is not available for the OCL parser to validate. Table 5 summarizes these uncorrected errors.

This determines our study with 840 invariants which were successfully parsed and analyzed. The rest of the research questions deal exclusively with this set of parsed invariants. It is also worth noting that the metamodels UML Composite Structures (uml6) and UML Components (uml10) do not contain any parsed invariant after the preprocessing phase, so the final number of analyzed metamodels is 31 instead of 33.

6.1.2. Q2. How balanced is the distribution of OCL invariants definitions in the domain structure?

In order to answer this question, we look at the proportion of classes in the domain structure that serve as the context for the invariants. Each line in table 6 displays the cumulated proportion of invariants defined on each percentile of domain structure classes. Every proportion is rounded to two decimal points. For example, for the OCL Types metamodel ("ocl1"), 17% of the invariants were defined on 10% of the classes; 30% of invariants are defined on 20% of its classes, and so on. In metamodels where the domain structure is not big enough to calculate a subset of metaclasses with a given percentile, a dash ('-') is given. The table is sorted from the most balanced metamodel (top line) to the most unbalanced. We observe that 23 out of 31 metamodels define their complete sets of invariants taking as context only

Metamodel	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%	55%	60%	65%	70%	75%	80%
dc	-	-	-	-	0,33	0,33	0,33	0,33	0,33	0,67	0,67	0,67	0,67	0,67	1	1
ocl1	-	0,17	0,30	0,30	0,43	0,52	0,52	0,61	0,70	0,78	0,78	0,87	0,91	0,91	0,96	1
rbac	-	0,27	0,27	0,52	0,52	0,67	0,67	0,79	0,79	0,85	0,91	0,91	0,97	0,97	1	1
erre	-	0,20	0,37	0,47	0,54	0,61	0,68	0,75	0,80	0,85	0,85	0,88	0,92	0,93	0,95	0,97
dwf	0,13	0,30	0,39	0,48	0,57	0,65	0,74	0,83	0,91	1	1	1	1	1	1	1
ocl2	0,12	0,29	0,40	0,50	0,60	0,69	0,74	0,79	0,83	0,88	0,90	0,95	1	1	1	1
mtep	-	0,30	0,48	0,67	0,81	0,93	0,96	1	1	1	1	1	1	1	1	1
cspf	-	0,59	0,70	0,81	0,85	0,89	0,93	0,96	1	1	1	1	1	1	1	1
sam	0,35	0,58	0,72	0,79	0,87	0,90	0,93	0,96	0,97	1	1	1	1	1	1	1
cor1	0,33	0,50	0,67	0,79	0,88	0,96	1	1	1	1	1	1	1	1	1	1
hrc	0,33	0,56	0,70	0,82	0,92	1	1	1	1	1	1	1	1	1	1	1
cmof	0,45	0,65	0,75	0,83	0,93	1	1	1	1	1	1	1	1	1	1	1
emof	0,44	0,67	0,79	0,85	0,94	1	1	1	1	1	1	1	1	1	1	1
cor3	-	-	-	1	1	1	1	1	1	1	1	1	1	1	1	1
uml13	-	0,40	0,60	0,80	1	1	1	1	1	1	1	1	1	1	1	1
uml11	0,25	0,63	0,75	1	1	1	1	1	1	1	1	1	1	1	1	1
cor2	0,38	0,50	0,75	0,88	1	1	1	1	1	1	1	1	1	1	1	1
dg	0,38	0,77	0,92	1	1	1	1	1	1	1	1	1	1	1	1	1
uml12	0,44	0,73	0,88	1	1	1	1	1	1	1	1	1	1	1	1	1
b	0,44	0,78	1	1	1	1	1	1	1	1	1	1	1	1	1	1
xms	0,45	0,74	0,90	1	1	1	1	1	1	1	1	1	1	1	1	1
cwm	0,57	0,77	0,91	1	1	1	1	1	1	1	1	1	1	1	1	1
uml9	0,62	0,98	1	1	1	1	1	1	1	1	1	1	1	1	1	1
sad3	0,63	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
uml1	0,63	0,93	1	1	1	1	1	1	1	1	1	1	1	1	1	1
uml5	0,68	0,95	1	1	1	1	1	1	1	1	1	1	1	1	1	1
uml2	0,83	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
uml3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
uml4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
uml7	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
uml8	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 6: Proportion of invariants in subsets of classes.

40% of classes from the domain structure. An interesting observation is found in column of percentile 25%; here we see that 25 metamodels define 80% or more of their invariants in a subset of only one quarter of classes of the domain structure.

Nevertheless, the table makes it clear that our sample of metamodels comprises both balanced and unbalanced metamodels. In the first case, we can see for the RBAC metamodel how its integral set of invariants is spread across 75% of classes of its domain structure. In the case of this domain, a dynamic security approach, a large majority of concepts need to be specified with invariants. In this version of this metamodel, three quarters of its concepts have associated invariants expressing rules for correct models of RBAC. On the other end, the SAD3 metamodel defines its whole set of invariants on only 10% of classes on its domain structure. However, when looking back at figure 11, we realize that the specified set of invariants is very small, forcing its invariants to be defined on a small subset of classes. In the case of another unbalanced metamodel, UML State Machines (uml9), shows a case where invariants are spread across the domain structure but there is a concentration in a small subset of classes. This is because there are clearly concepts in the domain structure that carry a higher significance in the metamodel. In this case, the classes State, Transition, FinalState and Pseudostate are used as context of 26 invariants out of 42 defined in this metamodel. It is also noted, however, that the domain structure of this metamodel imports a large number of concepts of the UML infrastructure, which are not directly related to the State Machines domain but enlarge however the domain structure.

6.1.3. Q3. How pervasive are OCL invariants in the domain structure?

Figure 12 displays the distribution of invariant complexities to the domain structure for each metamodel. For example, among the 42 invariants of UML State Machines (uml9) the least complex invariants use only one element (complexity 1) from the domain structure and the most complex ones use seven (complexity 7).

We observe that among 25 metamodels out of 31, their complexity varies between 1 and 8. Twelve metamodels define invariants with a complexity in the range of 1 to 4. Of all metamodels, the ER to RE transformation metamodel shows a special case of very complex invariants, the highest measure being of 38; this is due to the special purpose of the OCL expressions in this metamodel, which were constructed to specify the outcome of a model

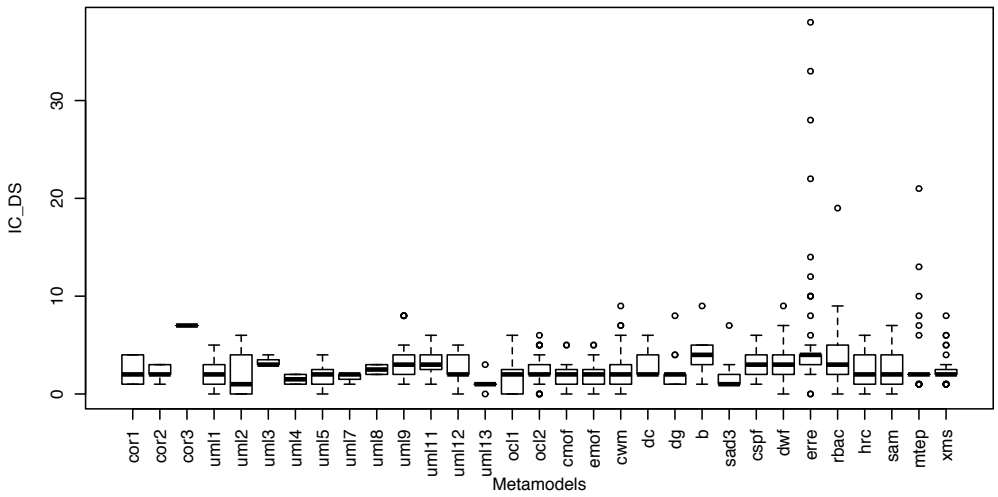


Figure 12: Boxplot for the measure of IC_DS across all metamodels.

transformation. Also we can notice the specific case of UML State machines which defines two invariants of complexity 7 (0.84% of the analyzed invariants). Among all invariants, 87.62% have a complexity of 4 or below. This means that this percentage of the studied OCL expressions contain references to 4 or less model elements of the domain structure.

Figure 13 provides another perspective on the complexity of invariants with respect to the UML State Machines metamodel. In this *treemap*, each class c of the metamodel is represented as rectangle, the area of the rectangle is proportional to the size of $c.classContent()$ and the gray level corresponds to the $c.CC\_DS$ (lighter for invariants of the class that use few elements of the domain structure, darker for invariants defined over many elements). Following the discussion from the preceding question regarding the fact of a few classes concentrating a big number of invariants, here we observe that few classes define invariants that strongly couple them to the rest of the domain structure. For example, Pseudostate and State define invariants which use 11 and 9 properties, respectively. These invariants belong also to the subset we identified in preceding question to carry a big part of the set of invariants. On the other hand, ProtocolConformance, Port, TimeEvent and Vertex present a class complexity of 0, either because their invariants do not invoke properties

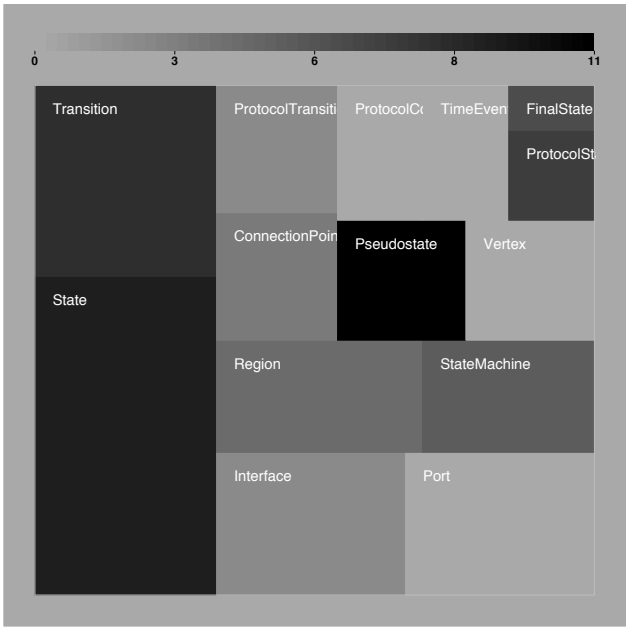


Figure 13: Treemap measuring CC_DS for the classes of the UML States Machine modeling language.

directly or because they do not define invariants. Invariants may also present a complexity of 0 when their body consists of the invocation on an operation on the same context (there is no direct invocation of properties).

We can emphasize two general observations about question 3: (i) there are strong variations in invariants complexities from one metamodel to the other, even if most of them (25/31) define simple invariants (complexity with respect to the domain structure (IC_DS) is lower than 8); (ii) when analyzing metamodels with complex invariants, it appears that there is also a strong variation in invariant complexities among classes and that few metaclasses concentrate the most complex invariants.

6.1.4. Q4. What is the extent of the usage of the OCL language constructs in metamodels?

Figure 14 shows the occurrence frequency of the different OCL expression types in the analyzable invariants defined within the metamodels under study. For example, in 840 invariants under study, we find 3423 occurrences of the OperationCallExp expression from OCL. The navigation expressions OperationCallExp, VariableExp, PropertyCallExp are the most present ex-

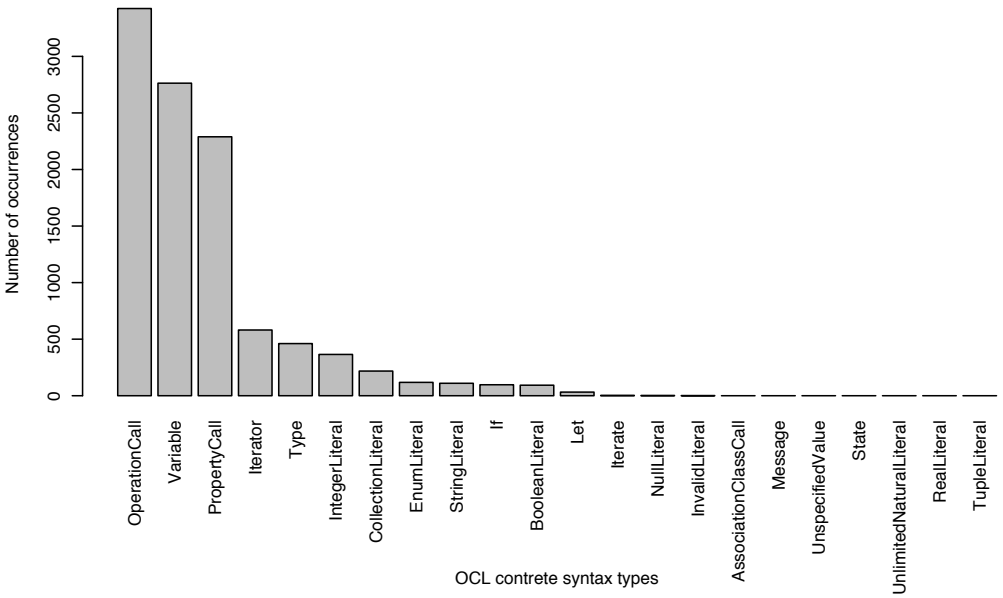


Figure 14: Frequency of OCL constructs in analyzed invariants.

pressions in the invariants we analyze. When adding the types TypeExp, IteratorExp, CollectionLiteralExp, EnumLiteralExp, BooleanLiteralExp, IfExp and IntegerLiteralExp we capture 98.60% of the OCL constructs present in the 840 invariants. Furthermore, 96.90% of these invariants rely only on these constructs, whereas only 3.1% make some use of the remaining expression types. This means that 96.90% of valid invariants in OMG specifications are expressed with 45.45% of the OCL (10 constructs out of 22 concrete expression types).

This unbalanced use of the OCL language might indicate several things. The low number of occurrences of string literals (StringLiteralExp), 22, might suggest a guideline to not use strings in invariant definitions. The low usage of 'if' expressions (IfExp) seems consistent with the previous observations of low IC_DS values for most invariants. Since 'if' expressions require at least three subexpressions (condition, 'then' and 'else'), it is very unlikely to find an invariant using 'if' with a complexity lower than 2. One particular type of expression, OperationCallExp, deserves special attention since due to its nature of an expression used to invoke an operation, and different operations are called among the matched occurrences. Figure 15 shows the frequency of the called operations. The most important observation comes from the fact

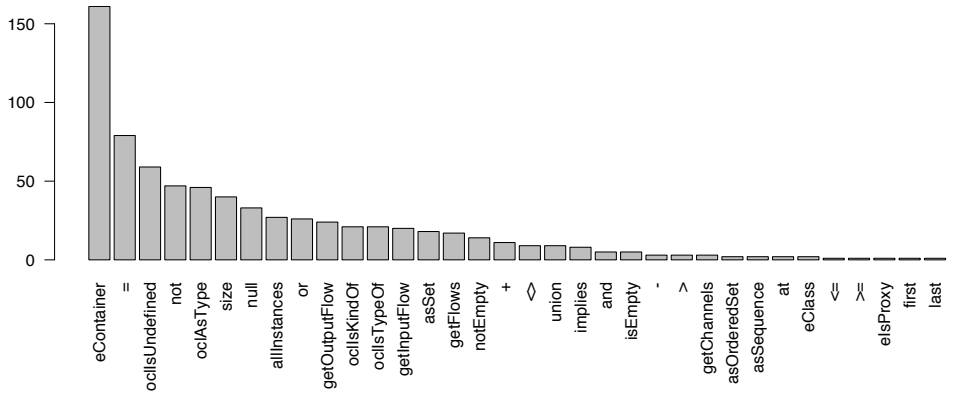


Figure 15: Operation called in the found instances of OperationCallExp.

that the most invoked operation, *eContainer()*, does not belong to the official OCL specification. It is a helper operation defined by some OCL interpreters, among which Topcased, which allows to navigate the composition relationships, by being called on the composed object. This is perhaps an indication that more versatile operations to navigate through the different types of relationships between object are needed in the OCL specification. The figure also shows some frequent operations such as getInputFlow() or getOutputFlow(), are ad-hoc operations added to certain metamodels, in this case UML, by the domain expert to ease the expression of well-formedness rules.

From another perspective, figure 16 shows a *treemap* for the CC_OCL metric on the UML State Machines classes. We observe that the usage of the OCL is different from one class to the other. For instance, the invariants in Region use the largest number of OCL expression types, which correspond to the 8 types mentioned above.

6.2. Threats to validity

Our study was conducted as accurately as possible, given the inputs specified in section 4, with the aforementioned assumptions. Nevertheless, we identify here possible construction, internal and external threats to validity.

Internal threats lie on the source of the empirical data. For the industrial and academic groups, the main source of data was a call for participation in the PlanetMDE mailing list. As the premier mailing list of practitioners from industry and academia in the Model-Driven Engineering world, we can assert the representativeness of this population. Furthermore, in these groups the

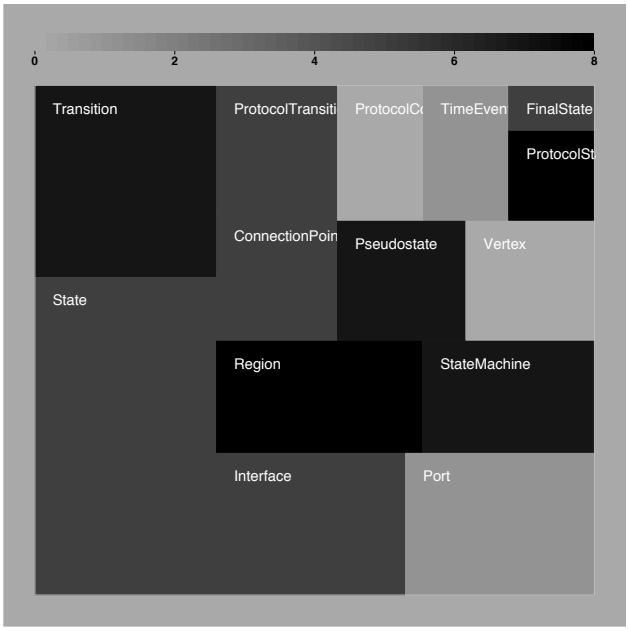


Figure 16: Treemap measuring CC_OCL for the classes of the UML States Machine modeling language.

developers constructing the metamodel can present levels of expertise ranging from beginner to expert, and thus this might influence greatly on the quality of the metamodels. This is less threatening in the case of the standards group, as all standards come from the OMG, the organization that created the MOF and OCL themselves, and furthermore accounts for great experience in the creation of domain models [26]. For the standards group, we have examined three OMG specifications based on the availability of machine-readable files. In some cases we manually edited these files to be able to process them by our tools; this manual step is prone to errors. Likewise, the process of fixing constraints was performed respecting the intentions of the specification writers, albeit it remains a human process subject to errors. Since we seized metamodels available from the web, we have no pointer about the skills of the developers who have written the invariants. It is possible that well trained modelers could write more complex invariants, or use a larger portion of the OCL.

Construction threats lie in the way we define our metrics and their measurement. The way we have defined metrics to answer our research questions responds to our own judgement on how to measure this phenomena. How-

ever, another choice of metrics and statistical descriptive analysis may yield different results and consequently produce different conclusions. Validity of our results could also be affected by analysis and calculations performed by our program OCL Metrics. Although the algorithms were designed to follow precisely the metric definitions presented in section 3, and integrally analyzed the abstract syntax tree of each invariant, subtleties and assumptions made by the underlying tools (particularly Eclipse OCL) are the subject of possible bias. Our metrics result might be too coarse grained to draw pertinent conclusions, and other metrics might be better fitted for this purpose.

External threats lie on the statistical significance of our study. In industry group, we only have 3 metamodels; however they contribute a total of 172 invariants to our base. We acknowledge that we have only observed 840 syntactically valid invariants. We do not know to what extent this can be generalized to invariants that define languages from other domains.

7. Related Work

To our knowledge, there has not been another study on the articulated usage of MOF and OCL for metamodeling. On the OCL side, a very important effort has been made by Gogolla et al. [11] when analyzing different OCL environments (both parsing and checking), to find the different implementations that have been made of the standard. Although an important contribution that motivated our research question about the usage of OCL constructs, the study does not go into surveying practices in modeling or metamodeling.

Our work emerges from a need to better understand metamodeling practices. We focus on the conjunct usage of OCL and MOF, but there are many other activities for metamodeling. For example, some works explain processes to build a metamodel that generalizes a set of existing metamodels in a given domain. Beydoun et al. [27] discuss the mix of top-down and bottom-up process they have followed to build a generic metamodel for multi-agent systems, starting from a set of existing metamodels in this domain. Monperrus et al. [28] define a systematic process to build a requirements metamodel with an explicit measurement purpose.

In this study we have considered metamodels defined with implementation of OMG standards, such as Ecore for Essential MOF and USE for UML. However, it should be noted that other implementations of these standards also exist, such as the Generic Metamodeling Environment (GME) [29]

which is also based in MOF. Another popular metamodeling environment is MetaEdit+, a commercial tool that provides an integrated tool suite to define metamodels and automatically generate end-user model editors [30].

Regarding the definition, formalization and implementation of metrics on models, extensive work exists in the field of metrics for UML modeling as an activity in object-oriented analysis and design. The goal is to assess model quality, either at the model level or the metamodel level.

At the model level, Gronback provides a list of metrics and recommended value ranges to ensure model quality, called "audits" [31]. In future work, we plan to establish a set of guidelines based on the metrics presented in our study, that assist the application of the best practices for metamodeling. Lange et al. [32] focus on the quality of software development processes relying on UML models, and as such they propose a set of metrics on modeling artifacts. It is interesting how some of these metrics, such as the count of model elements and class complexity have a relationship to metrics in our study, namely SDS and IC_DS respectively. In earlier work of the same authors, they perform empirical analysis on a sample UML models, and propose a quantitative measuring of the completeness of a software design with UML models. They assess diagram well-formedness and completeness, and inter-diagram consistency [33].

At the metamodel level, the OMG has proposed the Structured Metrics Metamodel [34], and Monperrus et al. [35] propose an approach for the definition of metrics at the meta level, associated to a generative approach, which provides a measurement tool on models. It is completely model-driven, so the definition of metrics is a model itself that is coupled to the user's metamodel, and allows the automatic generation of a measurement tool to be executed in the user's models. Another work at the metamodel level, Hein et al. [36] propose a set of generic metrics written in OCL which are evaluated on the user's metamodel.

It should be noted though that the motivation of all these works is model quality, and as such the metrics suggested by these works are mostly an adaptation of the metrics of the Object-Oriented Programming world adapted to models. However, there exists no metrics approaches with the goal of measuring and understanding the usage of languages such as MOF and OCL forming complex structures. Furthermore, there exists no approach regarding the usage of two articulated formalisms. Nevertheless, McQuillan et al. [37] discuss the challenges in definition and implementation of metrics across different viewpoints throughout different abstraction levels of a software sys-

tem. This was our case when creating metrics for the different views of a metamodel, namely the object-oriented structure and the logic-based well-formedness rules.

Metrics about usage of language constructs have also been developed for empirical studies in software engineering, albeit not in the model-driven engineering world. For instance in [38] the authors focus on language grammars, and explore different proposals of metrics to measure the quality and complexity of these grammars. It is also worth noting Muñoz et al. [39], where the authors measure the usage of features offered by aspect-oriented programming languages in open source projects.

8. Conclusion

Model-driven engineering encourages domain experts to embody their knowledge in the form of a metamodel. This metamodel can serve to define the valid structure of all models in the domain. However, experts who wish to precisely specify the scope of their domain have to master two different formalisms for metamodeling: an object-oriented, MOF-compliant, language to model the domain structure and a logic-based language to add rules that further specify the structure of models. The conjunct usage of two languages for metamodeling represents a major challenge, which is not currently supported by methodologies nor best practices.

The intuition of this work is that a systematic observation of practices in different areas can provide hints on how these two languages are used together. We have performed an empirical enquiry of the conjunct usage of OCL and MOF in 995 invariants over 33 metamodels. We have made available this collection of data to provide the community with an openly available benchmark to carry further experiments on MOF and OCL. Our analysis was based on a new set of metrics, which reveal various aspects of the coupling and scattering of OCL rules in the metamodels. We formally defined these metrics and embedded them in an extensible tool that automates the analysis over metamodels stored with different formats.

We observed that domain experts tend to identify a small set of essential concepts in their domain structure in the context of which they express most well-formedness rules. We also observed that well-formedness rules are loosely coupled to the metamodel, *i.e.* most of the rules are defined over less than 5 concepts of the domain structure. Despite this low coupling, we also observed that the usage of two languages hinders maintenance tasks in

metamodeling: 155 OCL invariants out of 995 could not be analyzed because they did not match with the MOF structure anymore. Although OCL became the *de facto* formalism to express well-formedness rules over MOF-compliant structures, this is not the initial intent of the language. Consequently, we observe that a significant portion of the language is never used in well-formedness rules: 10 out of 22 constructs of the concrete syntax were never used to define our observed set of invariants.

This survey indicates that the conjunct use of OCL and MOF is a difficult task and that experts are more or less likely to master OCL's logic for precise metamodeling. Based on our findings, we propose the following actionable outputs for practitioners:

- Our main advice we propose for metamodeling stakeholders is the throughout checking of metamodel specifications, containing both metamodel and well formedness rules, with the help of a syntax checking tool, validating both the correct usage of the OCL language syntax and the syntactically correct usage of the modeling elements found in the underlying domain structure.

- Our set of MOF-OCL metrics can be used for comparative analysis for the expert's well-formedness rules. One could be interested, for instance, in comparing two sets of metrics where one contains a majority of expressions defined for a specific metaclass as their context, and the other where their contexts are scattered throughout the metamodel, and evaluate how choosing one of these two approaches affects the complexity with respect to the domain structure, making them more or less understandable for users of the metamodel, and so on.

- We realize that OCL is a very rich language, oftentimes exceeding the needs of writing well formedness rules. We have identified the effective subset of the language that new practitioners unfamiliar with OCL are required to learn in order to express these rules. Likewise, a supporting tool focused only on this subset would greatly simplify work for these practitioners.

Future Work. As a next step for this analysis we plan to look for patterns in the usage of OCL for metamodel invariants. Recurring patterns could be used to assist the development of new metamodels and provide concrete guidelines for precise metamodeling. Such guidelines could help

mature the capitalization of knowledge in a metamodel similarly to the work of Mernik et al. for domain-specific programming languages [40]. It is equally important to explore the topic of metamodel reusability. Since reuse is a main concern in model-driven engineering, we need to assess the question whether well-formedness rules improve or hinder the reusability of a model, what issues could arise and how to tackle them by means of our tooling solution proposed in this work.

9. Acknowledgements

The authors wish to thank Jens Brüning, Bastien Coatanéa-Gouachet, Pierre Gaufillet, Prof. Martin Gogolla, Akram Idani, Mirco Kuhlmann, Holger Schmidt, for their valuable contribution to our study with data subjects. We also thank James Hill, Bran Selic, Juha-Pekka Tolvanen and Fabian Buttner for their support about this study. Special thanks to Véronique Thelen, associate professor in the Economy department of University of Lille 1, for her helpful advice on our quantitative analysis.

[1] G. Edwards, N. Medvidovic, A methodology and framework for creating domain-specific development infrastructures, in: 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2008, pp. 168–177.

[2] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, S. Völkel, Design guidelines for domain specific languages, in: 9th OOPSLA Workshop on Domain-Specific Modeling, 2009, pp. 7–13.

[3] R. B. France, B. Rumpe, Model-driven development of complex software: A research roadmap, in: Future of Software Engineering, 2007, pp. 37–54.

[4] OMG, Meta Object Facility Core, v2.0 (2006).

[5] OMG, Object Constraint Language, v2.2 (2010).

[6] J. Cadavid, B. Baudry, H. Sahraoui, Searching the boundaries of a modeling space to test metamodels, in: 5th IEEE International Conference on Software Testing, Verification and Validation (ICST), Montréal, Canada, 2012.

[7] J. Cadavid, B. Baudry, B. Combemale, Empirical evaluation of the conjunct use of mof and ocl, in: Experiences and Empirical Studies in Software Modelling (EESSMod 2011), CEUR, 2011.

[8] B. Selic, Uml 2 specification issue 6462, http://www.omg.org/issues/issue6462.txt, updates dating until 2008. (2003).

[9] B. Combemale, X. Crégut, P.-L. Garoche, X. Thirioux, Essay on Semantics Definition in MDE. An Instrumented Approach for Model Verification, Journal of Software 4 (9) (2009) 943–958.

[10] M. Gogolla, J. Bohling, M. Richters, Validating UML and OCL models in USE by automatic snapshot generation, Software and Systems Modeling 4 (4).

[11] M. Gogolla, M. Kuhlmann, F. Büttner, A benchmark for ocl engine accuracy, determinateness, and efficiency, in: 11th International Conference on Model Driven Engineering Languages and Systems (MODELS), 2008, pp. 446–459.

[12] J. Warmer, A. Kleppe, The object constraint language: getting your models ready for MDA, Addison-Wesley, 2003.

[13] A. L. Baroni, Quantitative assessment of uml dynamic models, SIGSOFT Softw. Eng. Notes 30 (2005) 366–369.

[14] D. Kolovos, R. Paige, F. Polack, On the evolution of OCL for capturing structural constraints in modelling languages, in: Dagstuhl Workshop on Rigorous Methods for Software Construction and Analysis, 2009.

[15] ACM, Some shortcomings of OCL, the object constraint language of UML.

[16] OMG, Unified Modeling Language, v2.2 (2009).

[17] F. Budinsky, E. Merks, D. Steinberg, Eclipse Modeling Framework, 2nd Edition, Addison-Wesley Professional, 2009.

[18] Eclipse uml2 (2014).
URL http://www.eclipse.org/modeling/mdt/?project=uml2

[19] Eclipse ocl (2014).
URL http://www.eclipse.org/modeling/mdt/?project=ocl

[20] OMG, UML Profile For CORBA And CORBA Components, v1.0 (2008).

[21] Sourceforge corba project (2014).
URL http://sourceforge.net/projects/eclipsecorba/

[22] OMG, Diagram Definition v1.1 beta 2 (2010).

[23] OMG, Common Warehouse Metamodel v1.1 (2010).

[24] P. Muller, F. Fleurey, J. Jézéquel, Weaving executability into object-oriented meta-languages, 8th International Conference Model Driven Engineering Languages and Systems (MODELS) (2005) 264–278.

[25] S. Sen, N. Moha, B. Baudry, J. Jézéquel, Meta-model pruning, 12th International Conference on Model Driven Engineering Languages and Systems (MODELS) (2009) 32–46.

[26] ACM, Specifications, not meta-models.

[27] G. Beydoun, G. Low, B. Henderson-Sellers, H. Mouratidis, J. Gomez-Sanz, J. Pavon, C. Gonzalez-Perez, Faml: A generic metamodel for mas development, IEEE Transactions on Software Engineering 35 (6) (2009) 841 –863.

[28] M. Monperrus, J.-M. Jézéquel, B. Baudry, J. Champeau, B. Hoeltzener, Automated measurement of models of requirements, Software Quality Journal.

[29] M. Emerson, J. Sztipanovits, T. Bapty, A mof-based metamodeling environment, Journal of Universal Computer Science 10 (10) (2004) 1357–1382.

[30] J. Tolvanen, S. Kelly, Metaedit+: defining and using integrated domain-specific modeling languages, in: 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications (OOPSLA), ACM, 2009, pp. 819–820.

[31] R. Gronback, Model validation: Applying audits and metrics to uml models, in: Borland Developer Conference, 2004.

[32] C. Lange, M. Chaudron, Managing model quality in uml-based software development, in: 13th IEEE International Workshop on Software Technology and Engineering Practice, 2005., IEEE, 2005, pp. 7–16.

[33] C. Lange, M. Chaudron, An empirical assessment of completeness in uml designs, in: Proceedings of the 8th International Conference on Empirical Assessment in Software Engineering (EASE '04), 2004, pp. 111–121.

[34] OMG, Structured Metrics Meta-Model, v1.0 (2012).

[35] M. Monperrus, J. Jézéquel, B. Baudry, J. Champeau, B. Hoeltzener, Model-driven generative development of measurement software, Software and Systems Modeling (2010) 1–16.

[36] C. Hein, M. Engelhardt, T. Ritter, M. Wagner, Generation of formal model metrics for mof based domain specific languages, Electronic Communications of the EASST.

[37] J. McQuillan, J. Power, On the application of software metrics to uml models, Models in Software Engineering (2007) 217–226.

[38] M. Črepinšek, T. Kosar, M. Mernik, J. Cervelle, R. Forax, G. Roussel, On automata and language based grammar metrics, Computer Science and Information Systems/ComSIS 7 (2) (2010) 309–329.

[39] F. Muñoz, B. Baudry, R. Delamare, Y. Le Traon, Inquiring the usage of aspect-oriented programming: an empirical study, in: 25th IEEE International Conference on Software Maintenance (ICSM), IEEE, 2009.

[40] M. Mernik, J. Heering, A. M. Sloane, When and how to develop domain-specific languages, ACM Comput. Surv. 37 (4) (2005) 316–344.