STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIV**, Number 2, 2009

PROPOSAL OF A SET OF OCL WFRS FOR THE ECORE META-METAMODEL

VLADIELA PETRAŞCU, DAN CHIOREAN, AND DRAGOŞ PETRAŞCU

Abstract. Specifying a complete set of OCL Well Formedness Rules (WFRs) is essential for the well-definedness of any (meta)modeling language's abstract syntax. Within this paper, we report on the definition of a set of OCL WFRs for the Ecore meta-metamodel. We use some relevant WFRs for the Ecore generics accompanied by meaningful test cases, in order to illustrate our proposal and its advantages over related approaches.

1. Introduction

The highest level of abstraction in any model-driven approach, be it Model Driven Architecture (MDA) [5], Model Driven Engineering (MDE) [14], or Language Driven Development (LDD) [8], is represented by the metamodeling language - the language used for defining all modeling languages, itself included. This is level M3 of the classical four-level modeling architecture promoted by the Object Management Group (OMG). It is well known that a complete definition of any language should include formal representations of its *abstract syntax*, *concrete syntax*, and *semantics*; for modeling languages, the abstract syntax is generally given in terms of a metamodel (which describes the concepts used by the language and their relationships), which should be accompanied by appropriate Well Formedness Rules (WFRs, that further constrain the legal instantiations of metamodel concepts). It follows that the abstract syntax of a metamodeling language should be defined by means of its meta-metamodel and associated WFRs. OMG's Meta Object Facility (MOF) [3], Eclipse Modeling Framework's (EMF's) Ecore [15], and eXecutable Metamodelling Facility's (XMF's) XCore [8] are probably the best known meta-metamodels today.

Received by the editors: November 1, 2009.

2010 *Mathematics Subject Classification.* 68N30.

1998 *CR Categories and Descriptors.* D.2.4 [**SOFTWARE ENGINEERING**]: Software/Program Verification – *Programming by contract, Class invariants, Validation*; D.2.11 [**SOFTWARE ENGINEERING**]: Software Architectures – *Languages (e.g., description, interconnection, definition)* .

Key words and phrases. Model Driven Engineering (MDE), meta-metamodel, Object Constraint Language (OCL), Well Formedness Rules (WFRs), Ecore.

Having a complete set of metamodel WFRs is of utmost importance for the well-definedness of any modeling language. This is due to the fact that the graphical formalism used to represent the metamodel itself (that of class diagrams) is not powerful enough so as to capture all the constraints that govern the individual meta-concepts, as well as their inter-relationships. Even more, these WFRs should be formalized, in order to allow the existing tools to validate the models against them; a model is considered to be valid/correct if and only if it conforms to both its metamodel and associated WFRs. The language used in order to formalize the WFRs should be the Object Constraint Language (OCL) [7], for the following three reasons at least. First of all, OCL is the standard formalism. Second, defining the WFRs as OCL invariants is preferred to implementing them directly in the repository code, since the OCL expressions are more compact and intelligible compared to their equivalents in a programming language. Third, today we can benefit from a powerful tool support, regarding both the evaluation of OCL constraints on snapshots and their automatic translation into code. The Object Constraint Language Environment (OCLE) [6] and EMF [1] with Model Development Tools (MDT)-OCL [4] are notable examples in this respect.

The arguments above are even stronger when it comes to metamodeling languages. Their abstract syntax is used in defining the metamodels of all possible modeling languages. There has to be possible to check/ensure the correctness of all these metamodels which are to be reused by instantiation in thousands of modeling applications.

Still, a study that we have carried out on the three above mentioned meta-metamodels, MOF, Ecore, and XCore, has revealed that the goal of having a correct and complete set of OCL WFRs for each of them is far from being reached. The closest to this aim is Ecore, whose repository code contains a set of WFRs implemented directly in Java. In case of the OMG MOF standard, a great number of OCL specifications used in describing the Core UML Infrastructure (which is part of MOF) are wrong. As for XCore, it only contains two such WFRs, specified using the OCL-like language XOCL.

Given this state of facts, our overall aim has been to define a complete set of OCL WFRs for each of the three meta-metamodels, as well as to test and validate them on relevant metamodel examples (finding appropriate test models even before or simultaneously with defining the OCL constraints - test driven (meta(-meta))modeling - is highly important). Within this context, the current paper reports on the definition and validation of such a complete set of OCL WFRs for Ecore.

The rest of the paper is organized as follows. Section 2 reviews the state of facts regarding the Ecore WFRs, which provides the context and motivation of our work. Section 3 testifies our contribution, using some relevant WFR for

the Ecore generics. Related work is summarized in Section 4. We conclude the paper in Section 5, giving some hints on future work.

2. Ecore WFRs. State of Facts

Compared to the other meta-metamodels that we have studied, namely MOF and XCore, Ecore has a special status, that may be described by the following:

- Ecore is, beyond any doubt, the best known EMOF (Essential MOF) implementation. However, Ecore does not match EMOF exactly. On the one side, the approach taken with Ecore is more pragmatic and implementation-oriented. On the other side, starting with EMF 2.3, Ecore includes constructs for modeling with generics [11]; this is considered to be a departure from EMOF, which does not currently provide such support.
- Due to the framework it ships with (EMF), Ecore is definitely the most tested meta-metamodel.
- The Ecore repository includes a set of WFRs that allow validating the metamodels that instantiate it. These rules are implemented within the EcoreValidator class. However, although the code does contain comments, these do not reflect all implementation decisions. Especially in case of those rules used to check the correctness of parameterized types, the code complexity is increased and the lack of detailed comments and examples is disturbing. The fact that, as stated in [11], "The design of Ecore's support for generics closely mirrors that of Java itself" is expected to help in this respect. Still, the tests that we have run have shown that there are differences among the two, regarding both the declaration of generic types and their correct instantiation.
- The Ecore implementation witnesses the fact that the value of meta-metamodel level WFRs has been acknowledged. However, even though EMF integrates an OCL plugin (MDT-OCL), we have not found any OCL equivalent of the implemented constraints.
- The paper [9] proposes some OCL WFRs that may be used in validating the Ecore generics; this is actually the only paper concerning the OCL formalization of Ecore WFRs that we have found in the literature. However, even though they are a good starting point and comparison base, the OCL specifications described there are far from complete and not entirely correct.

Within this context, our goal has been to identify, classify and specify in OCL a complete set of WFRs for Ecore, complete at least with respect to the rules already implemented in EMF. We report on accomplishing this goal in the following sections using some relevant constraints concerning generics.

The OCL specifications are preceded by their informal equivalents and are accompanied by relevant test cases used for their validation.

3. A Complete Set of OCL WFRs for Ecore

Taking the EMF implementation as a reference, we have defined a complete set of OCL WFRs for Ecore, which we have tested and validated on relevant examples using OCLE. The entire set can be found at [2]. Within this section, we detail on one such WFR, regarding the Ecore generics. Choosing this particular constraint for exemplification purposes is due to both its complexity level (since it is a non-trivial WFR) and the fact that it allows a close comparison with related work described in [9].

3.1. **Generics in Ecore.** Figure 1 shows that part of the Ecore metametamodel that ensures the generic modeling support it provides. As previously mentioned, this has been introduced starting with EMF 2.3, the newly added concepts being `ETypeParameter` and `EGenericType`. We briefly explain and exemplify these concepts in the following.

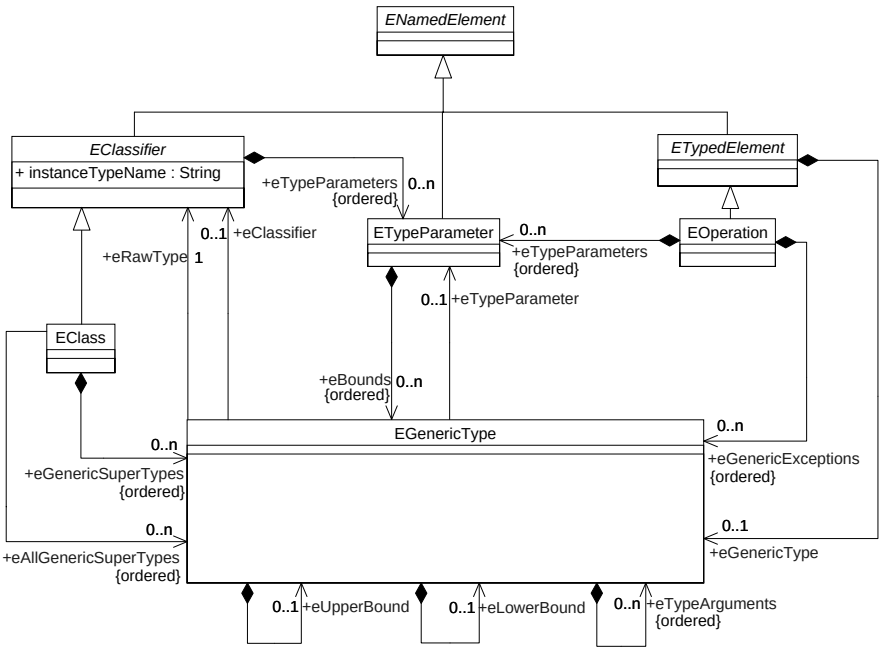


Figure 1. Ecore generics

Similar to Java, Ecore supports generic type and operation declarations, as well as generic type instantiations (also known as parameterized types).

An ETypeParameter instance stands for a type parameter used by either a generic classifier or a generic operation declaration. That particular ETypeParameter is contained by its corresponding EClassifier or EOperation instance. This is denoted by the composition relationships EClassifier - ETypeParameter and EOperation - ETypeParameter from Figure 1, which are mutually exclusive (xor constraint). As an example, the Java generic type declaration interface Collection<T> would be modeled in Ecore by means of an EClass instance named Collection, having its interface attribute set to true, and whose eTypeParameters sequence contains a single ETypeParameter instance, named T.

Type parameters may have bounds, as indicated by the composition relationship between ETypeParameter and EGenericType. For a Java type declaration such as

```
1 class OrderedList<T extends Comparable<T>> { ... }
```

the eBounds sequence owned by the type parameter T contains a single EGenericType instance, namely Comparable<T>.

An EGenericType instance may denote one of the following: a type parameter reference, a (generic) type invocation, or a wildcard. This is reflected by its associations to ETypeParameter and EClassifier, respectively. The two associations are mutually exclusive; there is a WFR specifying that an EGenericType instance cannot be simultaneously associated to both an eTypeParameter and an eClassifier. In case it has an eTypeParameter, then it is a type parameter reference, if it has an eClassifier, then it is a (generic) type invocation, and when both are missing, it is a wildcard. An EGenericType instance denoting a generic type invocation may specify type arguments (see the eTypeArguments role name); in case it does not specify any type arguments, then it is used as a raw type, the reason being that of ensuring compatibility with the previous, non-generic EMF releases. Wildcards may specify a lower or an upper bound (see the corresponding unary compositions of EGenericType). To exemplify all these, let us consider the following Java interface definition:

```
2 interface List<T> extends Collection<T>
3 {
4   boolean add(T elem);
5   boolean addAll(Collection<? extends T> col);
6   ...
7 }
```

The listing above contains a generic type declaration for List. In the equivalent Ecore model (Figure 2), this would be modeled by means of an EClass instance named List, which contains an ETypeParameter instance with the

name `T`. The newly declared type specifies a generic supertype, `Collection<T>`. The latter is modeled using an `EGenericType` instance that corresponds to a generic type invocation with a type argument; the referred classifier is `Collection` and the contained type argument `T`. At its turn, this type argument is an `EGenericType` instance that corresponds to a type parameter reference, the referenced type parameter being the `ETypeParameter` instance `T`. The fourth line of the listing contains another `EGenericType` instance that corresponds to a type parameter reference, only this time it is used not as a type argument, but as the type of the `EParameter` instance `elem`. The type of `col` in line 5 denotes a generic type invocation; the referenced classifier is again `Collection` and the type argument is `? extends T`. The latter is an `EGenericType` instance that corresponds to an upper bounded wildcard; it has no `eClassifier` or `eTypeParameter` and it specifies an `eUpperBound`, namely `T`.

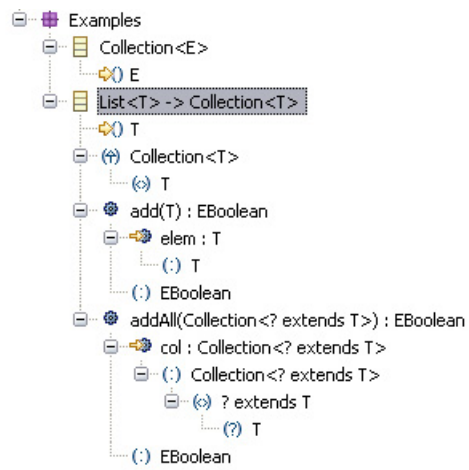


FIGURE 2. Ecore model for `List<T>`. EMF tree-editor screenshot

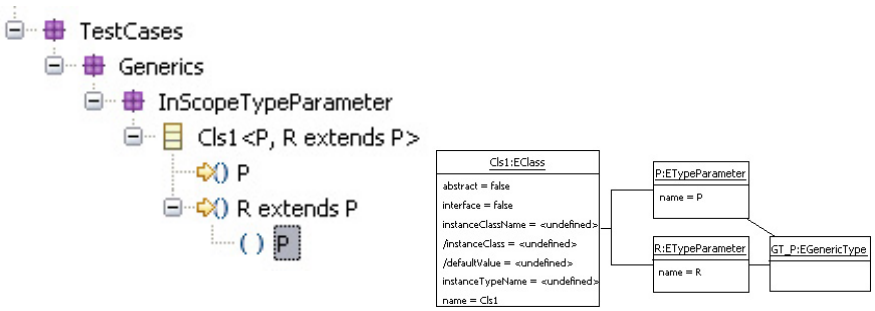
`EGenericType` instances can play various roles in an Ecore model, each kind of usage being constrained by suitable WFRs. Such an instance can be exactly one of the following:

1. A generic supertype of a class, as shown by the composition relationship `EClass-EGenericType`; `Collection<T>` in line 2 of the listing above is such an example;
2. The type of a typed element (attribute, reference, operation, parameter), as shown by the composition relationship between `ETypedElement` and `EGenericType`; an example is `T` in line 4 above;
3. A bound of a type parameter, as shown by the composition relationship `ETypeParameter-EGenericType`; `Comparable<T>` in line 1 above is such an example;
4. One of the type arguments of a generic type invocation, fact denoted by the unary composition relationship of `EGenericType` owning the `eTypeArguments` role; `T` from `Collection<T>` in line 2 above is a good example;
5. The upper or lower bound of a wildcard, as shown by the other two unary compositions of `EGenericType`; `T` from `Collection<? extends T>` in line 5 above is an example of an upper bound usage of a generic type;

6. An exception type, fact denoted by the composition between `EOperation` and `EGenericType`.

3.2. **A WFR for Generics in Ecore.** Equipped with this knowledge regarding the Ecore generics, we seek to provide an OCL specification for the following informal WFR: "Assuming that a generic type denotes a type parameter reference, the referenced type parameter must be in scope and must not be a forward reference. The type parameter is in scope if its container is an ancestor of this generic type within the corresponding Ecore containment tree". We give a few examples in the following, in order to ensure a deeper understanding of the rule and to set up some test cases for its validation. Assuming a closer familiarity of the reader with Java than Ecore, we start with the Java equivalent of each chosen example, followed by OCLE and EMF snapshots for the corresponding Ecore model.

As a first example, let us consider the Java declaration `class Cls1<P, R extends P>`. The generic type declaration for `Cls1` uses `P` and `R` as type parameters, the latter being upper bounded by the former. This is a valid generic declaration since the referenced type parameter `P` is in scope and is not a forward reference (`P` being declared prior to `R`). The equivalent Ecore model consists of an `EClass` instance named `Cls1` which contains two `ETypeParameter` instances named `P` and `R` (Figure 3 shows the corresponding EMF and OCLE snapshots). The type parameter `R` has a bound, which is an `EGenericType` instance that references the type parameter `T`. The OCLE snapshot shows explicitly the `EGenericType` instance used as a bound (`GT_P`) and its link to the referenced type parameter. Within the EMF tree, the bound appears as a direct descendent of the type parameter it bounds, being labeled with the name of the referenced type parameter.



(A) EMF snapshot (B) OCLE snapshot

FIGURE 3. Ecore model for Example 1: `Cls1<P, R extends P>`

For the second example, consider the Java declarations `class Cls2<Q>` and `class Cls3<S, T extends Q>`. The second one is obviously not valid, since the bound of `T` references type parameter `Q`, which is out of scope. The corresponding OCLE snapshot is shown in Figure 4; its EMF equivalent is missing since the framework constrains a referenced type parameter to be chosen from the list of those in scope. Therefore, it is impossible to model such a case using the EMF tree-like editor. However, this erroneous situation could still occur if the model were loaded from an XMI file instead of being created directly with the editor.

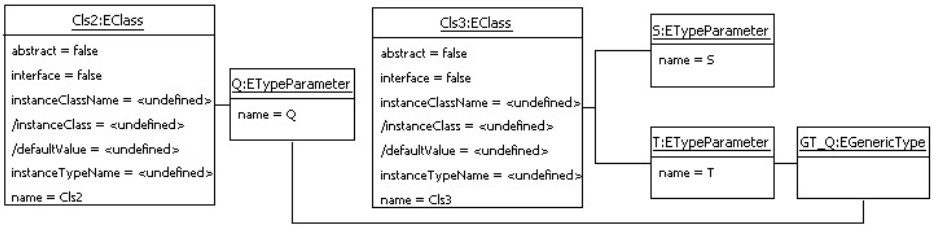
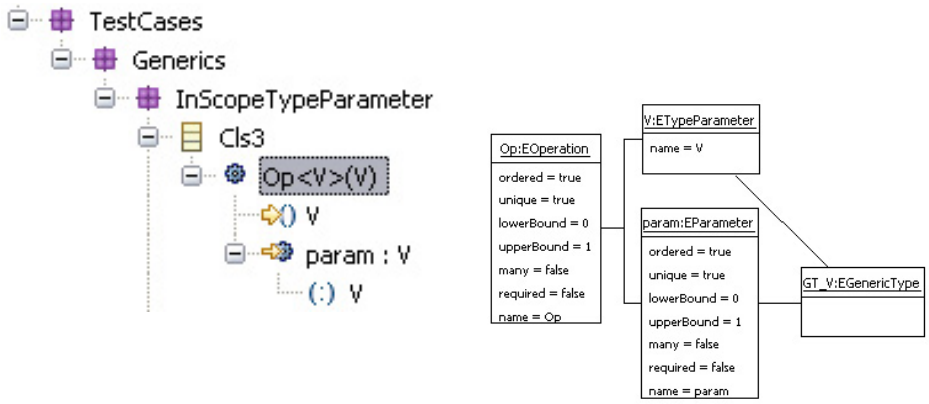


FIGURE 4. OCLE snaphot for Example 2: `Cls2<Q>, Cls3<S, T extends Q>`

In both examples described above, the container of each type parameter has been a classifier. Let us now consider the following Java generic operation declaration `<V> void Op(V param)`. The equivalent Ecore model has at its root an `EOperation` instance, `Op`, whose `eTypeParameters` sequence contains only the type parameter `V`. `Op` owns a single parameter, `param`, whose type is an `EGenericType` instance that references the type parameter `V`. The EMF and OCLE snapshots are illustrated in Figure 5. This is again a valid model with respect to the WFR under consideration.

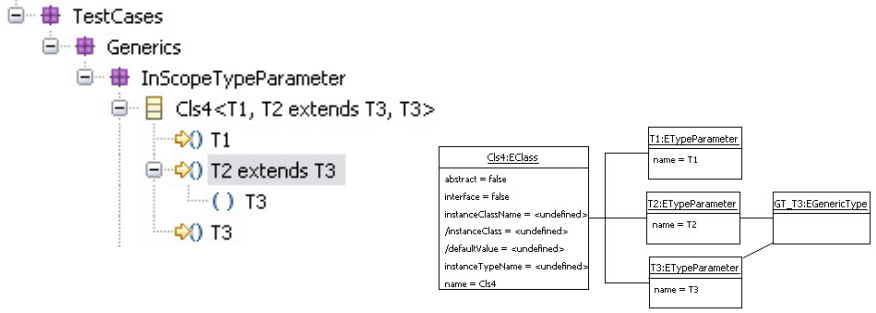
The fourth and last example we take is again a generic class declaration, of the form `class Cls4<T1, T2 extends T3, T3>`. Such a declaration is not valid, since the bound of `T2` performs a forward referencing of the type parameter `T3`. The equivalent snapshots are given in Figure 6.

The constraints in Listing 1 formalize the WFR stated at the beginning of this subsection. The OCL specification has been splitted in two invariants defined for the `EGenericType` context, namely `InScopeTypeParameter` and `NotForwardReference`; as their names indicate, the former enforces the type parameter referenced by a generic type to be in scope, while the latter checks for forward referencing. As in programming, the splitting of large constraints into smaller pieces is a good modeling practice. This way, the constraints become easier to write and their comprehensibility is enhanced. Even more, this also provides valuable support in localizing exactly and in real time the cause of a constraint violation during model checking activities.



(A) EMF snapshot (B) OCLE snapshot

FIGURE 5. Ecore model for Example 3: `<V> void Op(V param)`



(A) EMF snapshot (B) OCLE snapshot

FIGURE 6. Ecore model for Example 4: `Cls4<T1, T2 extends T3, T3>`

```
1  context EGenericType
2   -- The referenced type parameter must be in scope, i.e.,
3   -- its container must be an ancestor of this generic type ...
4   inv InScopeTypeParameter:
5    self.isTypeParameterReference() implies
6    self.ancestors()->includes(self.eTypeParameter.eContainer())

8  context EGenericType
9   -- ... and must not be a forward reference.
10  inv NotForwardReference:
11   (self.isTypeParameterReference() and self.isUsedInATypeParameterBound())
12   implies
```

```
13   (let refParameter : ETypeParameter = self.eTypeParameter
14    let boundedParameter : ETypeParameter = self.boundedTypeParameter()
15    let paramSeq:Sequence(ETypeParameter)=
16      (if refParameter.eContainer().oclIsKindOf(EClassifier)
17       then refParameter.eContainer().oclAsType(EClassifier).eTypeParameters
18       else refParameter.eContainer().oclAsType(EOperation).eTypeParameters
19       endif)
20    let posRefParameter : Integer = paramSeq->indexOf(refParameter)
21    let posBoundedParameter : Integer =
22      (if paramSeq->includes(boundedParameter)
23       then paramSeq->indexOf(boundedParameter)
24       else -1
25       endif)
26    in
27    ( (posBoundedParameter <> -1) implies
28      ( (posRefParameter < posBoundedParameter) or
29        ( (posRefParameter = posBoundedParameter) and
30          (not boundedParameter.eBounds->includes(self))
31        )
32      )
33    )
34   )
```

LISTING 1. `EGenericType` invariants prohibiting invalid type parameter references

We do not insist on the OCL specification for `InScopeTypeParameter` (lines 1 to 6 of Listing 1), since it carefully matches the comments it goes along with. However, we detail the three query operations it makes use of, namely `isTypeParameterReference()`, `eContainer()` and `ancestors()`. Their OCL definitions are provided in Listing 2.

The core query here is `ancestors()`, which should compute all parents of an arbitrary object from within the Ecore containment tree to which the object belongs. The returned set should include the object's direct container, the direct container of the latter, and so on. In case the particular object is the root of the tree, then the empty set is returned. In order to provide its intended functionality, `ancestors()` makes use of `eContainer()`, which returns the direct container of an arbitrary object.

The `eContainer()` operation is given a default definition for the root of the Ecore modeling hierarchy, `EObject`, which is overriden in all its descendants, according to the composition relationships they are involved in (see Figure 7). The default implementation returns `Undefined(EObject)` (Listing 2, lines 12-13) and most of the overridings simply perform a one-step navigation of a composition relationship (Listing 2, lines 52-65). However, the composition relationships which involve `ETypeParameter` and `EGenericType` are uni-directional in the Ecore model, therefore the OCL expressions for

`eContainer()` in these two particular cases (Listing 2, lines 15-50) are more complex and unefficient, due to the calls to `allInstances()`.

```
1  context EGenericType
2   def: isTypeParameterReference() : Boolean =
3         not self.eTypeParameter.isUndefined()

5  context EObject
6   def: ancestors() : Set(EObject) =
7        let empty : Set(EObject) = Set{} in
8        if self.eContainer().isUndefined() then empty
9        else Set{self.eContainer()}->union(self.eContainer().ancestors())
10       endif

12 context EObject
13  def: eContainer() : EObject = oclUndefined(EObject)

15 context EGenericType
16  def: eContainer() : EObject =
17   let cls=EClass.allInstances->any(c|c.eGenericSuperTypes->includes(self))
18   let param=ETypeParameter.allInstances()->any(p|p.eBounds->includes(self))
19   let te=ETypedElement.allInstances()->any(t|t.eGenericType=self)
20   let gt1=EGenericType.allInstances()->any(g|
21             g.eTypeArguments->includes(self))
22   let gt2=EGenericType.allInstances()->any(g|g.eLowerBound=self)
23   let gt3=EGenericType.allInstances()->any(g |g.eUpperBound=self)
24   let op=EOperation.allInstances()->any(o|
25            o.eGenericExceptions->includes(self))
26   in
27   (if not cls.isUndefined() then cls
28    else if not param.isUndefined() then param
29         else if not te.isUndefined() then te
30              else if not gt1.isUndefined() then gt1
31                   else if not gt2.isUndefined() then gt2
32                        else if not gt3.isUndefined() then gt3
33                             else if not op.isUndefined() then op
34                                  else oclUndefined(EObject)
35                                  endif
36                             endif
37                        endif
38                   endif
39              endif
40         endif
41   endif)

43 context ETypeParameter
44  def: eContainer() : EObject =
45   let classifier = EClassifier.allInstances()->any(c |
46                      c.eTypeParameters->includes(self))
47   in
48   (if not classifier.isUndefined() then classifier
49    else EOperation.allInstances()->any(o | o.eTypeParameters->includes(self))
50    endif)
```

```
52 context EPackage
53  def: eContainer() : EObject = self.eSuperPackage

55 context EClassifier
56  def: eContainer() : EObject = self.ePackage

58 context EStructuralFeature
59  def: eContainer() : EObject = self.eContainingClass

61 context EOperation
62  def: eContainer() : EObject = self.eContainingClass

64 context EParameter
65  def: eContainer() : EObject = self.eOperation
```

LISTING 2. Query operations used by `InScopeTypeParameter`

`ETypeParameter`, for instance, is involved in two composition relationships (with `EClassifier` and `EOperation`, see Figure 1 for reference), both of which are unidirectional, navigable only from container to part (we did not manage to find the rationale for this design decision). Therefore, the direct container of a type parameter is always either a classifier or an operation. However, this container cannot be accessed through a simple navigation. The only way to identify it involves searching that particular type parameter within the `eTypeParameters` collections of all classifiers and operations that belong to the current model (which explains the use of `allInstances()` in lines 45, 49 of Listing 2). That operation or classifier which includes the searched type parameter within its `eTypeParameters` collection is its direct container. There will definitely be at most one such classifier or operation, since the considered relationships are compositions. Therefore, the use of the undeterministic `any()` in lines 45, 49 of Listing 2 is completely safe; it should produce the same result no matter the tool used. The overriding of `eContainer()` for `EGenericType` can be justified in a similar manner, only this time the number of composition relationships involved, therefore the complexity, is greater.

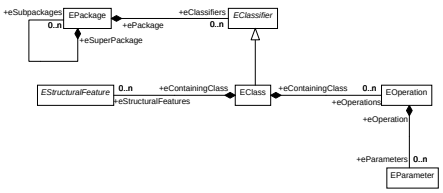


FIGURE 7. Ecore containment relationships

The evaluation of `InScopeTypeParameter` using OCLE has ended successfully for the first, third and fourth of the test examples considered above,

while failing for the second, in accordance with the results given by the EMF EcoreValidator and the Java compiler. In case of the first test example, the constraint is evaluated for the `EGenericType` instance `GT_P`, which references the `ETypeParameter` instance `P`. The computed set of `ancestors()` of `GT_P` contains the `ETypeParameter` instance `R` (its direct container, of which it is a bound) and the `EClass` instance `Cls1` (the direct container of `R`). Also, the direct container (`eContainer()`) of `P` is `Cls1`. Since the latter belongs to the `ancestors()` set, the constraint evaluates to true on `GT_P`. The evaluation results for the second and fourth examples (`false` and `true`, respectively) can be explained in a similar way. In case of the third one, the invariant is evaluated on the `EGenericType` instance `GT_V`, which references type parameter `V`. The `ancestors()` of `GT_V` are its direct container, `param`, (`GT_V` being the `eGenericType` of `param`) and the `EOperation` instance `Op` (the direct container of `param`); the `eContainer()` of `V` is `Op`, therefore the required inclusion takes place, which ends successfully the evaluation.

In order to simplify the reading, we will use the phrase "generic type" instead of "`EGenericType` instance" from here on.

The invariant that completes the proposed WFR's OCL definition, `NotForwardReference`, rules out all generic types which reference type parameters that are in scope, but are declared afterwards. This situation can only occur when the generic type is contained in a type parameter bound, either of a classifier or of an operation. The employed query operations, specifically `isUsedInATypeParameterBound()` and `boundedTypeParameter()`, are defined in Listing 3.

```
1  context EGenericType
2   def: isUsedInATypeParameterBound() : Boolean =
3    -- checks whether self is involved in defining a type parameter bound
4    self.ancestors()->exists(o | o.oclIsTypeOf(ETypeParameter))

6  context EGenericType
7   def: boundedTypeParameter() : ETypeParameter =
8    -- returns the type parameter in whose bound self is used
9    self.ancestors()->any (o |
10     o.oclIsTypeOf(ETypeParameter)).oclAsType(ETypeParameter)
```

LISTING 3. Query operations used by `NotForwardReference`

A generic type is said to be used in a type parameter bound if and only if there is an `ETypeParameter` instance among its ancestors. This is expressed by means of line 4 of Listing 3 above. If that particular `ETypeParameter` instance is its direct container, then the generic type identifies itself with the bound, otherwise it is contained at a certain level in this bound. If a generic type is involved in defining the bound of a type parameter, then this will be the only `ETypeParameter` instance among its ancestors (since no containment,

direct or not, is possible among type parameters). Therefore, the use of `any()` within the OCL expression which returns the type parameter in whose bound the current generic type is involved (line 9, Listing 3) is safe.

Resuming to the OCL definition of `NotForwardReference` (lines 8-34, Listing 1), we should make clear that a forward reference can only happen when a generic type, let us call it `GT`, references a type parameter, let us call it `T`, which is declared at a later time compared to the moment of use of `GT`. Since type parameters can only be declared within a generic classifier or operation definition, and assuming that the referenced type parameter is in scope (out of scope type parameters are ruled out by the first invariant), it follows that `GT` can only be involved in defining a bound for a type parameter owned by the same classifier or operation that owns `T`. This explains line 11 from the definition of `NotForwardReference`. Following this, the invariant computes the referenced type parameter (`refParameter`), the bounded type parameter (`boundedParameter`), and the sequence of all parameters owned by the direct container of `refParameter` (`paramSeq`). For the invariant to evaluate to `true`, `refParameter` should be declared prior to `boundedParameter` in `paramSeq` (line 28, Listing 1), or the two should be the same type parameter (line 29, same listing). In the latter case, however, it is prohibited for a type parameter to bound itself (line 30). Therefore, a situation such as the following `class Cls5<P1, P2 extends P2>` is not allowed, since `P2` bounds itself. Still, a declaration of the kind `Cls6<P3, P4 extends Cls<P4>>`, in which `P4` is involved in defining its own bound, is valid.

From the test examples above, the one intended to capture forward referencing was the fourth. There, the `NotForwardReference` invariant will be evaluated for the generic type `GT_T3`, which bounds type parameter `T2` and references type parameter `T3`. The sequence of all parameters having the same container as the referenced one evaluates to `Seq{T1,T2,T3}`, from which it is obvious that the position of the referenced type parameter (3) is greater than the one of the bounded parameter (2). Therefore, the boolean expression in lines 28-32 of Listing 1 evaluates to `false`, and so does the whole invariant.

4. Related work

As already mentioned in Section 2, the only benchmarks we have for comparing our work with are the EMF implementation of the EcoreValidator and the paper [9].

4.1. **The EMF EcoreValidator.** We have already made clear in Section 1 which are the advantages derived from using OCL, instead of a programming language, in formalizing WFRs. In addition, in Section 2, we have pointed out some of the drawbacks of the current EMF implementation of Ecore's WFRs. Among them, there have been mentioned some discrepancies between

the Java specification of generics and the corresponding WFRs implemented by the EMF EcoreValidator. We will take one such example in the following, so as to justify a new OCL WFR that we propose for the Ecore generics and that should also be implemented by the EcoreValidator.

Concerning the correct declaration of generic types and methods, the Java Language Specification [10] (pp. 50) states the following constraints: "Type variables have an optional bound, T & I_1 ... I_n. The bound consists of either a type variable, or a class or interface type T possibly followed by further interface types I_1, ..., I_n. ... It is a compile-time error if any of the types I_1 ... I_n is a class type or type variable. The order of types in a bound is only significant in that ... and that a class type or type variable may only appear in the first position."

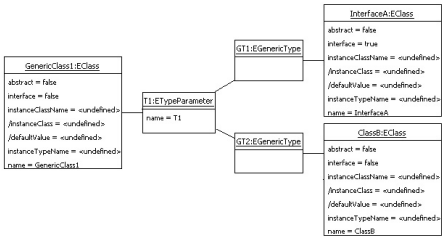


FIGURE 8. OCLE snaphot corresponding to `GenericClass1 <T1 extends InterfaceA & ClassB>`

Therefore, a generic type declaration of the kind

```
1 class GenericClass1 <T1 extends InterfaceA & ClassB>
```

where `InterfaceA` is an interface type and `ClassB` is a class type, gives the following compile-time error in a Java environment "The type ClassB is not an interface; it cannot be specified as a bounded parameter". However, by modeling the exact same type in EMF and validating it, the validation completes successfully.

In a similar manner, the declaration

```
2 class GenericClass2 <T1, T2 extends T1 & InterfaceA>
```

generates the Java compile-time error "Cannot specify any additional bound InterfaceA when first bound is a type parameter", while its equivalent Ecore model validates successfully under EMF.

This is due to the fact that the `EcoreValidator` class does not include code for checking the above mentioned constraints. Therefore, we propose the following OCL WFR for the Ecore generics:

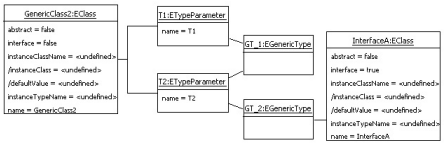


FIGURE 9. OCLE snaphot corresponding to `GenericClass2 <T1, T2 extends T1 & InterfaceA>`

```
1  context ETypeParameter
2   inv ValidBounds:
3    -- If a type parameter has bounds and the first bound is a type parameter
4    -- reference, then there are no other bounds.
5    (self.eBounds->notEmpty() and
6     self.eBounds->first().isTypeParameterReference()
7     implies self.eBounds->size() = 1
8    )
9    and
10   -- If there are at least two bounds,
11   -- then all except (maybe) the first one should refer to interface types.
12   (self.eBounds->size() >= 2
13    implies Sequence{2..self.eBounds->size()}->select(i |
14      not self.eBounds->at(i).hasInterfaceReference())->isEmpty()
15   )
```

LISTING 4. The `ValidBounds` OCL WFR

The aforementioned WFR makes use of the following query operations:

```
1  context EGenericType
2    def: hasClassifierReference() : Boolean =
3      not self.eClassifier.isUndefined()

5    def: hasClassReference() : Boolean =
6      self.hasClassifierReference() and self.eClassifier.oclIsTypeOf(EClass)

8    def: hasInterfaceReference() : Boolean =
9      self.hasClassReference() and self.eClassifier.oclAsType(EClass).interface

11   def: isTypeParameterReference() : Boolean =
12     not self.eTypeParameter.isUndefined
```

LISTING 5. Query operations used by `ValidBounds`

By evaluating the proposed WFR on the OCLE snapshots given in Figures 8 and 9, which correspond to the declarations in lines 1, respectively 2 above, the obtained result is `false` in both cases, in accordance with the Java specification.

4.2. **The approach taken in** [9]**.** In the following, we will focus on comparing our work with the one described in [9]. The above mentioned paper aims at stating a set of OCL constraints that allow checking whether (1) a given generic type declaration or (2) a corresponding instantiation with type arguments (a so called parameterized type) are well formed or not. For further reference and comparison, we provide in Listing 6 the OCL code for the `consistentTypeParameters` WFR, meant to accomplish the first goal above (the OCL specification used for accomplishing goal number 2 is omitted, since it is not directly comparable with the WFRs included in this paper). Its corresponding informal specification, as can be deduced from the paper, would be the following: "The type parameters of any classifier should have non-empty, distinct names. The bounds of a type parameter (if any) can reference either a type parameter or a classifier (they cannot be wildcards). If the bound references a type parameter, then the referenced parameter should be in scope; if it references a classifier, it should be a valid type invocation (either non-generic or generic, possibly raw)."

```
1  context EClassifier
2   inv consistentTypeParameters:
3    allDifferent(eTypeParameters.name) and
4    eTypeParameters->forAll(tp | tp.isConsistent(eTypeParameters))

6  context ETypeParameter::isConsistent(
7                                tpsInScope:Collection(ETypeParameter)):Boolean
8   def: self.name <> '' and (self.eBounds->isEmpty() or
9        self.eBounds->forAll(tr | tr.isConsistentTypeReference(tpsInScope)))

11 context EGenericType::isConsistentTypeReference(
12                           tpsInScope:Collection(ETypeParameter)):Boolean
13  def: not isWildcard() and
14       ( (self.isReferenceToTypeParameter() and
15          tpsInScope->includes(self.eTypeParameter))
16         xor
17         (self.isReferenceToClassifier() and
18          self.eClassifier.isValidTypeInvocation(self.eTypeArguments))
19       )

21 context EGenericType::isReferenceToTypeParameter():Boolean
22  def: eClassifier->isEmpty() and
23       not eTypeParameter->isEmpty() and eTypeArguments->isEmpty()

25 context EGenericType::isReferenceToClassifier():Boolean
26  def: not eClassifier->isEmpty() and eTypeParameter->isEmpty()
```

LISTING 6. The `consistentTypeParameters` constraint from [9]

The set of constraints described in [9] can be analysed with respect to both its declared purpose and our final goal of defining a complete set of WFRs for Ecore in general, and Ecore generics in particular.

Regarding the first criterion, there are certain shortcomings concerning these constraints, that we mention briefly in the following:

(1) The proposed constraints are incomplete with respect to their intended purpose. On the one side, those meant to check the well formedness of a generic type declaration only constrain the bounds of a type parameter to reference parameters from within the same type declaration, without prohibiting forward references (lines 14-15, Listing 6). However, forward referencing is not allowed, neither in EMF not in Java (whose generics' model inspired the one in Ecore). On the other side, for the WFRs that check the correct instantiation of a generic type definition (which are not reproduced here), only a skeleton is given. The OCL expressions for `captureConversion(...)` and `isSuperTypeOf(...)` (which are the core queries of these WFRs, both in matter of complexity and functionality) are missing from the paper and no reference to them is provided. As a consequence, it is impossible to evaluate on snapshots the correctness or efficiency of these WFRs.
(2) There is some redundancy in the OCL specification. The `isConsistentTypeReference(...)` query operation (starting in line 11 above) states that a generic type used in a parameter bound (1) should not be a wildcard and (2) shoud reference either a classifier or a type parameter. The latter implies the former, so it should be enough to only impose (2) as a constraint. Moreover, the `isConsistent(...)` operation (starting in line 6) requires any type parameter to have a nonempty name. However, in Ecore, `ETypeParameter` inherits `ENamedElement`, and the latter owns a WFR that checks the well formedness of its `name` attribute (well formed implies not empty).
(3) As a matter of style, the OCL specification patterns state that the use of `forAll` on collections should be avoided. The corresponding constraints should be rewritten to use either `reject` or `select`, thus allowing an easy identification of the cause of an evaluation-time error.

With respect to defining a complete set of WFRs for the Ecore generics, those described in [9] are only a small subset. They are only focused on the definition and instantiation of generic classifiers; generic operations are not taken into account. Moreover, even if a given generic type is a valid instantiation of a certain generic classifier, depending on its usage, it may be further constrained. There are various ways of using such a generic type (we have detailed on that in Subsection 3.1), with several constraints that result thereof. Here are some examples: a generic type used as a generic supertype should have a classifier that refers to a class; there may not be two different instantiations of the same generic classifier among the generic supertypes of a class; the classifier of a generic type that types an attribute should be a data

type instance, while the one used for a reference should be a class instance, and so on.

The WFRs that we have described here are part of a broader set of OCL WFRs meant to cover all constraints that apply to the Ecore concepts, generics included. Their expressions have been written following OCL specification patters that provide for an easy debugging in case of an evaluation-time error; therefore all usages of `forAll()` have been replaced with eqivalent `select()`/`reject()` rephrasings. Lines 13-14 of Listing 4 are a proof of this. Unefficient OCL constructs have only been used when there has been no other option (see the discussion related to the use of `allInstances()`), and the safety of using undeterministic constructs such as `any()` has been justified whenever the case. All WFRs have been tested and validated using OCLE.

The OCL WFRs that we have defined for generics take into account both generic classifier and generic operation declarations, as well as all previously mentioned usages of a generic type. From a completeness perspective, the WFR described in Subsection 3.2 is stronger than its equivalent part from Listing 6, since it checks for forward referencing, and this aligns it with both the EMF implementation and the Java specification; the one proposed in Subsection 4.1 is missing from the EMF implementation, while being enforced by the Java specification.

5. Conclusions and Future Work

In this paper we have reported on the definition of a set of OCL WFRs for Ecore. We have used medium-complexity WFRs for generics in order to illustrate our work and relate it to existing approaches in the literature. As far as we know, this is the first attempt to provide an OCL formalization of the well-formedness rules of Ecore. In a broader context, we have emphasized the importance of OCL WFRs in defining the abstract syntax of a (meta)modeling language.

Further work includes specifying complete sets of OCL WFRs for the other two meta-metamodels under study, MOF and XCore. This should lead to the identification of a "core" set of constraints used by all meta-metamodels. The definition of some relevant metamodels to help us in validating the proposed WFRs is also considered (in this respect, in [12], [13] we have already proposed and tested such a metamodel for components, named ContractCML).

Acknowledgements

This research has been realized in the framework of the IDEI research project "Frame based on the extensive use of metamodeling for the specification, implementation and validation of languages and applications", code

ID_2049, financed by the Romanian National University Research Council (CNCSIS).

References

[1] Eclipse Modeling Framework (EMF). http://www.eclipse.org/modeling/emf/.
[2] Frame Based on the Extensive Use of Metamodeling for the Specification, Implementation and Validation of Languages and Applications (EMF_SIVLA) project homepage. http://www.cs.ubbcluj.ro/∼chiorean/CUEM_SIVLA.
[3] Meta Object Facility (MOF) 2.0. http://www.omg.org/spec/MOF/2.0/.
[4] Model Development Tools (MDT) OCL. http://www.eclipse.org/modeling/mdt/?project=ocl.
[5] Model Driven Architecture (MDA). http://www.omg.org/mda/.
[6] Object Constraint Language Environment (OCLE). http://lci.cs.ubbcluj.ro/ocle/.
[7] Object Constraint Language (OCL) 2.0. http://www.omg.org/spec/OCL/2.0/.
[8] Tony Clark, Paul Sammut, and James Willans. *Applied Metamodeling. A Foundation for Language Driven Development.* Ceteva, 2008.
[9] Miguel Garcia. Rules for Type-checking of Parametric Polymorphism in EMF Generics. In Wolf-Gideon Bleek, Henning Schwentner, and Heinz Züllighoven, editors, *Software Engineering 2007 – Beiträge zu den Workshops*, volume 106 of *GI-Edition Lecture Notes in Informatics*, pages 261–270, 2007.
[10] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition.* Addison-Wesley Longman, May 2005.
[11] Ed Merks and Marcelo Paternostro. Modeling Generics with Ecore. In *EclipseCon 2007*, 5-8 March 2007. http://www.eclipsecon.org/2007/index.php.
[12] Vladiela Petraşcu, Dan Chiorean, and Dragoş Petraşcu. ContractCML - a Contract Aware Component Modeling Language. *SYNASC 2008 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 273–276, 2008.
[13] Vladiela Petraşcu, Dan Chiorean, and Dragoş Petraşcu. Component Models' Simulation in ContractCML. *Proceeding of Knowledge Engineering: Principles and Techniques (KEPT 2009), Studia. Universitatis Babes-Bolyai. Informatica, Special Issue*, pages 198–201, 2009.
[14] Douglas C. Schmidt. Model-Driven Engineering. *Computer*, 39(2):25–31, 2006.
[15] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley Professional, December 2008.

Babeş-Bolyai University, 1 Mihail Kogălniceanu, Cluj-Napoca, Romania
E-mail address: {vladi, chiorean, petrascu}@cs.ubbcluj.ro