Scalable Armies of Model Clones through Data Sharing

Erwan Bousse, Benoit Combemale, Benoit Baudry

Scalable Armies of Model Clones through Data Sharing

Erwan Bousse[1], Benoit Combemale[2], and Benoit Baudry[2]

[1] University of Rennes 1, France
erwan.bousse@irisa.fr
[2] Inria, France,
{benoit.combemale, benoit.baudry}@inria.fr

Abstract. Cloning a model is usually done by duplicating all its runtime objects into a new model. This approach leads to memory consumption problems for operations that create and manipulate large quantities of clones (e.g., design space exploration). We propose an original approach that exploits the fact that operations rarely modify a whole model. Given a set of immutable properties, our cloning approach determines the objects and fields that can be shared between the runtime representations of a model and its clones. Our generic cloning algorithm is parameterized with three strategies that establish a trade-off between memory savings and the ease of clone manipulation. We implemented the strategies within the Eclipse Modeling Framework (EMF) and evaluated memory footprints and computation overheads with 100 randomly generated metamodels and models. Results show a positive correlation between the proportion of shareable properties and memory savings, while the worst median overhead is 9,5% when manipulating the clones.

1 Introduction

Cloning a model consists in obtaining a new and independent model identical to the original one. An implementation of this operation can be found in the EcoreUtil.Copier class of the Eclipse Modeling Framework (EMF) [11], which consists in first creating a copy of the runtime representation of a model (*i.e.* the set of Java objects that represent the model) and then resolving all the references between these objects. Such an implementation is also known as *deep cloning*. This implementation is effective to produce valid, independent clones. However it has very poor memory performances for operations that require manipulating large quantities of clones (e.g. genetic algorithms [6], design space exploration [10] or model simulation traces [8]).

We address the performance limitations of current deep cloning operations by leveraging the following observation: given a metamodel and an operation defined for this metamodel, the operation usually writes only a subset of this metamodel. That means that it is possible to identify the *footprint* of the write accesses of these operations on a metamodel. This footprint is the set of *mutable* parts of the metamodel, *i.e.* elements that can be modified by an operation. The counterpart

of these elements, the *immutable* elements, are definitively stated at the creation of objects. Our intuition is the following: knowing the immutable elements of the metamodel, data could be shared between the runtime representation of a given model and its clones, saving memory when generating the clone.

In this paper, we propose a new *model cloning algorithm*, which implements different strategies to *share immutable data between clones*. This contribution relies on a specific runtime representation of the model and its clones in order to share the data and still provide an interface that supports the manipulation of the clones independently from each other. We articulate our proposal around the following questions:

- Considering that we know which parts of a metamodel are mutable, how can we avoid duplicating immutable runtime data among cloned models?
- Can it effectively save some memory at runtime when creating a high number of clones as compared to EMF cloning implementation ?

Our goal is both to give a solution that can be implemented in various existing execution environments, and to provide concrete evidence of the efficiency of such an approach on a widely used tool set: the Eclipse Modeling Framework (EMF). Section 2 motivates our problem. We present a list of requirements for cloning operators, and give the intuition of our idea regarding existing cloning techniques. Section 3 defines what we call model cloning and what are runtime representations of models. Section 4 presents the main contribution of this paper: a new approach for efficient model cloning. The idea is to determine which parts of a metamodel can be shared, and to rely on this information to share data between runtime representations of a model and its clones. We provide a generic algorithm that can be parameterized into three cloning operators (in addition to the reference *deep cloning* one): the first one only shares objects, the second only shares fields, and the third shares as much data as possible. Section 5 describes our evaluation, which was done using a custom benchmarking tool suite that relies on random metamodel and model generation. Our dataset is made of a hundred randomly generated metamodels and models, and results show that our approach can save memory as soon as there are immutable properties in metamodels. Finally, Section 6 concludes.

2 Motivation and Position

In this section we give requirements for cloning operators, and we explain how our idea is related to existing approaches

2.1 Requirements for Cloning

New activities have emerged in the model-driven engineering community in recent years, which all rely on the automatic production of large quantities of models and variations of models. For example, several works rely on evolutionnary computation to optimize a model with respect to a given objective [3, 6].

Optimization in this case, consists in generating large quantities of model variants through cloning, mutation and crossover and selecting the most fitted. In the field of executable domain specific modeling languages, modeling traces [8] (*i.e.* set of *snapshots* of the executed model) is a way to verify and validate the system through visualization or analysis of traces. Yet, a complete model trace consists in copying the state of the model at each simulation step, producing large quantities of model variants. Design space exploration [10] is the exploration of design alternatives before an implementation, which requires the generation of the complete design space (*i.e.* set of variations, which are models).

All these new MDE techniques produce *large* sets of models that originate from few models. From a model manipulation point of view, all these techniques require the ability to *clone*—possibly many times—an original model, and to query and modify the clones as models that conform to the same metamodel as the original. More precisely, we identify four requirements for model manipulation in these contexts

Req #1 scalability. Runtime representations of models must scale in memory.

Req #2 manipulation performance. It is necessary to manipulate the clones as efficiently as any model.

Req #3 model interface. The clones and the original model must be manipulated through the same interface.

Req #4 metamodel independence. Support model manipulation through a reflexive layer (the model operation is defined independently of a given metamodel).

Our work defines novel cloning operators that reduce the memory footprint of clones, while trying to comply with the aforementioned requirements. In particular, we evaluate the relevance of our solution with respect to the following four research questions:

RQ#1 Do the new operators reduce the memory footprint of clones, compared to deep cloning?

RQ#2 Can a clone be manipulated with the same efficiency as the original model?

RQ#3 Can a clone be manipulated using the same generated API as the original model?

RQ#4 Can a clone be manipulated using the reflective layer (e.g. as stated in the MOF Reflection package)?

2.2 Existing Cloning Approaches and Intuition

Object copying has existed since the beginning of object-oriented programming languages [4] with the *deep* and *shallow* copy operators. While the second operator does not ensure the independence of a clone and is thus not of interest, the first is at the basis of model deep cloning. Concerning models, the EMF provides a class named EcoreUtil.Copier with operations for deep copying sets of objects,

which can trivially be used to implement a model deep cloning operator. Yet, as stated previously, this operator does not fit our needs. In [5], Karsai et al. added model cloning to the Generic Modeling Environment (GME) in order to support model *prototyping*, *i.e.* applying the concepts of object prototyping [7] to models. However, this work considers that changes made in a model are reflected in its clones, whereas by definition a clone is independent from its origin. Overall, to our knowledge, no work attempted to tackle the requirements that we identified.

In terms of memory management, *copy-on-write* (a.k.a. lazy copy) is a widespread way to reduce memory consumption. The idea is the following: when a copy is made, nothing is concretely copied in memory and a link to the original element is created. At this point, both elements are identical, and accordingly reading the copy would in fact read the origin directly. But when writing operations are made on the copy, modified elements are effectively copied so that the copy keeps its own state and appears like a regular and independent element. Applied to model cloning, the runtime object configuration of a clone obtained using this technique would eventually only contain written mutable elements of the original model, which meets our need to reduce memory footprint (Req #1). However, it adds a considerable amount of control flow at runtime in order to detect when copies must be done, and such copies can happen unpredictably depending on the manipulations; this contradict the need for efficient clones (Req #2). More importantly, depending on the programming language used, this technique can be very difficult to implement; for instance, Java is pass-by-value, making it impossible to dynamically change the value of a variable from a different context (*i.e.* updating all references to an object that was just effectively copied), which is required to dynamically copy a model progressively and transparently.

Our intuition is that while deep cloning is easy to implement but memory expensive, and copy-on-write is memory-efficient but complicated with poorly efficient clones, it is possible to provide operators *in between* these two extremes. Similarly to the way copy-on-write discovers *dynamically* which parts of a model are mutable when copying written elements, our idea is to *statically* determine which elements that have to be copied at runtime. Such elements are opposed to the ones that can be referenced by both the original runtime representation and its clone. We present an approach based on this idea in the next section.

3 On Model Cloning

The purpose of this section is to clarify what we mean by the runtime representation of a model and to precisely define what we call a clone in this work.

3.1 Modeling

Since we focus on the runtime representation of models, we consider a metamodel to be the definition of a data structure. More precisely, we rely on the Meta-Object Facility (MOF) [9] that defines a metamodel as an object-oriented structure.

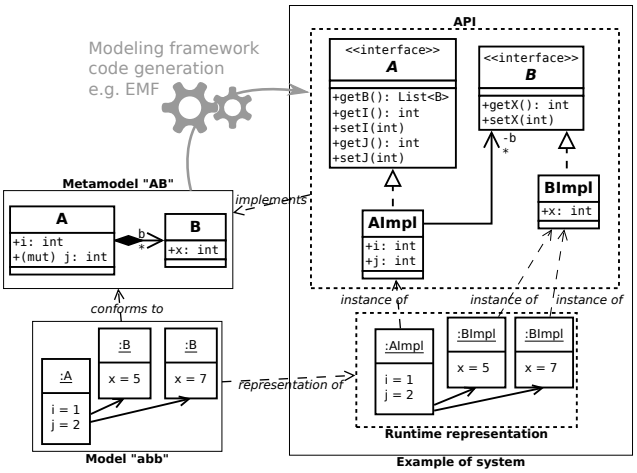


Fig. 1: Example of modeling and EMF usage with a sample metamodel `AB` and a sample model `abb`.

Definition 1. *A* metamodel *is an object oriented model defining a particular domain. More precisely, a metamodel is composed of classes composed of properties, a property being either an attribute (typed by a datatype) or a reference to another class. In practice, we consider a MOF model.*

Since a metamodel is composed of classes, a model that conforms to this metamodel is quite intuitively a set of objects that are instances of these classes.

Definition 2. *A* model *is a set of objects that conforms to a metamodel. Conformity means that each object in the model is an instance of one class defined in the metamodel. An* object *is composed of fields, each being based on a property of the corresponding class.*

During its lifecycle, a model can change in two possible ways: by creating/deleting objects or by changing values of fields of objects. We designate as *mutable* elements both the elements of a model that may change over time and the metamodel parts that define these elements. Our approach considers a given object configuration in order to produce a clone, and is thus not influenced by the creation of deletion of objects.

Definition 3. *A property of a class of a metamodel is* mutable *if, in each object instance of this class, the value of the field corresponding to this property can change after the construction of the object. Dually, a property is said to be* immutable *if its value cannot change after construction.*

Fig. 1 shows a metamodel named `AB` that is composed of two classes `A` and `B`. `A` has two attributes `i` and `j` and one reference `b`. `j` is mutable as specified by `(mut)`. `B` has a single attribute `x`. Below the metamodel, a model `abb` conforms

to AB and is composed of one object instance of A and two objects instance of B.

3.2 Implementation of Metamodels and Models

Specific execution environments are necessary to use metamodels and models. The Eclipse Modeling Framework (EMF) is one of the most popular. It generates Java interfaces and classes that implement a given metamodel, providing concrete mechanisms to create *runtime representations* of models that conform to the metamodel. We define a runtime representation as follows:

Definition 4. *The* runtime representation *of a model is the set of runtime data that is sufficient to reflect the model data structure. It must be manipulated through an interface that is consistent with the corresponding metamodel.*

Top right of Fig. 1 shows the API (Java interfaces and classes) generated by the EMF generator. Interfaces A and B define services corresponding to the data structure of the original metamodel AB, while Java classes AImpl and BImpl implement these interfaces. These elements support the instantiation and manipulation of runtime representations—here, Java object configurations—of models that conform to the metamodel. The bottom right of the figure shows a runtime representation of abb.

Note that a runtime representation that is eventually obtained using the EMF is structurally very similar to the original model: each object is represented by a Java object; each reference is represented by a Java reference; and each attribute is represented by a Java field. Yet runtime representations could theoretically take any form, as long as they are manipulated through an API that reflect the metamodel. One could imagine "empty" objects that get data from a centralized data storage component, or the use of a prototype-based programming language to create consistent runtime representations without defining classes.

3.3 Cloning

In this paper, we consider *cloning*[3] to be at the intersection of two main ideas: the exact duplication of elements and the independence of the obtained clone. Applied to models, a clone is therefore an independent duplication of some existing model. We define a clone as follows:

Definition 5. *A* clone *is a model that is, when created, identical to an existing model called the origin. Both models conform to the same metamodel and are independent from one to another*

Cloning a model is a deterministic procedure that has a unique possible output (*i.e.* a model identical to the original model). However there are multiple ways to implement this procedure for a given runtime environment. We therefore introduce the idea of *cloning operator* as follows:

[3] In terms of vocabulary, it is very similar to *copying*, and the choice of word is mostly a matter of habit. In this paper we rather *copy* objects and *clone* models.

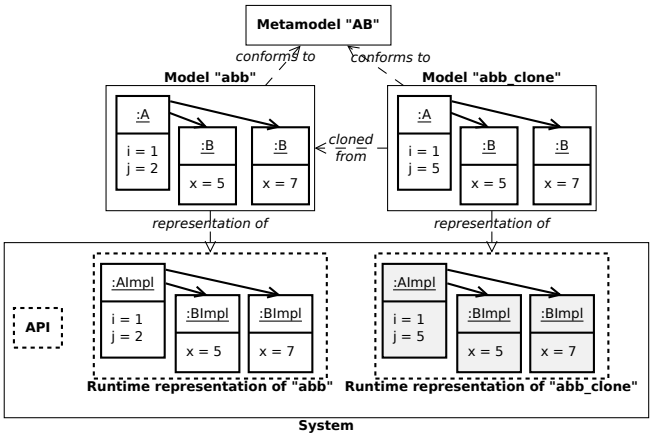


Fig. 2: Following Fig. 1, *deep cloning* of the model `abb`, which created a new model `abb_clone` along with a new runtime representation in memory. Then `abb_clone` diverged from `abb` by changing its `j` value.

Definition 6. *A* cloning operator *is an operator that takes the runtime representation of a model as input and returns the runtime representation of the clone of the model.*

Fig. 2 gives an example of cloning: the model `abb_clone` is a clone that was created at some point from the model `abb`. The moment the clone was created is important, since it is an independent model that can completely diverge from its origin; on this example, `abb_clone` already changed and has a different `j` value.

At the bottom right of Fig. 2, the runtime representation of `abb_clone` was obtained using the *deep cloning* operator. However, as stated in the previous section, runtime representations of models can virtually take any form, as long as it can be manipulated through an API consistent with the metamodel. This is what we investigate in the next section, where we present our main contribution: cloning operators that reduce the memory footprint of runtime representations of clones through data sharing.

4 Memory Efficient Cloning Operators

In this section we present our main contribution: an approach for memory efficient cloning through data sharing among runtime representations. For this work, we consider that input runtime representations were obtained using the EMF, *i.e.* each input runtime representation is identical to its model. Moreover, for our clones to be compliant with EMF, we ensure that each object of a clone is implemented by exactly one runtime object.

4.1 Data Sharing Strategies

When using the deep cloning operator, each object of a runtime representation is duplicated, which means twice as many objects and fields in memory. Our intuition is that since we know which parts of a metamodel are immutable, it must be possible to avoid duplicating some runtime objects and fields by safely using them for both the runtime representations of a model and its clones. Given a model conforming to a metamodel, we call *shareable* both the elements that can be shared between the runtime representations of the model and its clones, and the parts of the metamodel that define these elements.

In Section 2, we defined Req #2 (efficient manipulation of clones) and Req #4 (ability to define generic operations). However, sharing objects and fields between runtime representations necessarily breaks one or both of these requirements. First, if the same runtime object is shared between two runtime representations, it is supposed to represent two distinct objects—one per model. Therefore, it is possible for each of these objects to have a different *container*, since both objects are conceptually separate. The problem is that the MOF Reflection package states that each object must provide a `container()` operation that returns the unique container of an object, which is implemented in an operation of EMF `EObject` called `eContainer()`. Unfortunately, when a shared EMF runtime object is used, there is no way to know in which context (*i.e.* model) this manipulation occurs, and this operation thus cannot always return a unique container as expected. Therefore, generic operations that rely on this operation cannot be used on clones, which contradicts our Req #4. Second, we rely on a *proxy* design pattern to share the fields of runtime objects: a runtime object with a shareable field can be copied into a new runtime object without this field, but with a reference pointing to the original runtime object to provide access to this field. However, there is an overhead when accessing shared data through these proxy objects, which can be an issue with respect to Req #2.

Data sharing is essential to reduce the memory footprint of clones, which is our primary objective. Consequently, we designed several strategies that establish trade-offs between memory savings and satisfaction of Req #2 and Req #4. Modelers can then decide how to tune the cloning algorithm with respect to their specific needs. We provide four strategies that implement different interpretations of shareable metamodel elements:

DeepCloning Nothing is shareable.
ShareFieldsOnly Only immutable attributes are shareable.
ShareAll Shareable elements are immutable attributes, classes whose properties are all shareable, and immutable references pointing to shareable classes.
ShareObjOnly Same shareable classes as ShareAll, while properties are not.

If implementing the *DeepCloning* and *ShareFieldsOnly* strategies is quite straightforward, *ShareAll* and *ShareObjOnly* are more complicated because of a double recursion: shareable properties depend on shareable classes, and conversely. This can be solved using a fixed-point algorithm, or using the Tarjan

algorithm [12] to compute strongly connected components of a metamodel seen as a graph. We choose Tarjan in our implementation. Our approach to memory management through data sharing is quite close to the *flyweight* design pattern from Gamma et al. [2], which consists in identifying mostly immutable objects in order to share them between multiple objects. The main difference is that this pattern specifies that the mutable part of shared objects must be a parameter of all the operations of the objects, which contradicts our first requirement since the API of the clones hence differs from the one of the original model.

4.2 Generic Cloning Algorithm

Before defining our algorithms for model cloning, we introduce data structures and primitive functions on which the algorithms rely. We use pseudo-code inspired from prototype-based object-oriented programming [7], *i.e.* creating and manipulating objects without defining classes. The goal is to define the algorithms independently from any API that may be generated by a particular modeling framework. We consider the following structures and operations:

a runtime object o is created completely empty (*i.e.* no fields) using the *createEmptyObject*() operation. Fields can be added using *addField*(*name*,*value*), and can be retrieved using *getFields*().

a strategy is an object that implements one of the strategies given Section 4.1 with three operations:

- ***isFieldShareable***(f) returns true if, at the metamodel level, there is a shareable property represented by f.
- ***isObjShareable***(o) returns true if, at the metamodel level, the class of the object that match this runtime object is shareable.
- ***isObjPartShareable***(o) does the same, but for *partially shareable* classes, *i.e.* non-shareable classes with shareable properties.

copyObject(o) returns a copy of a runtime object o, i.e., a new object with the same fields and the same values. This is equivalent to the operation *copy* of EMF EcoreUtil.Copier

a runtime representation is a set of runtime objects. It can be created empty with *createEmptyRR*(), and it can be filled with objects using *addObject*(o).

a map is a data structure that contains a set of ⟨key,value⟩ pairs. It can be created with *createEmptyMap*() and be filled with *addKeyValue*(*key*, *value*).

resolveReferences (*map*) is an operation that, given a *map* whose keys and values are runtime objects, will create references in the values based on the references of the keys. This is equivalent to the operation *copyReferences* of EMF EcoreUtil.Copier.

The operation *copyObjectProxy*(o,*strategy*) is presented as Algorithm 1. It is parameterized by a strategy and an original object o, and it copies in a new object all the fields of o, except those considered shareable by the strategy. The last line of the operation creates a link to the original object in order to keep

Algorithm 1: *copyObjectProxy*

Data:
o, a runtime object
strategy, the strategy used (*i.e.* what is shareable)
Result: *p*, a proxy copy of *o*

```
1  begin
2      p ← createEmptyObject()
3      for f ∈ getFields(o) do
4          if ¬ strategy.isFieldShareable(f) then
5              p.addField(f.name, f.value)
6      p.addField("originObj", o)
```

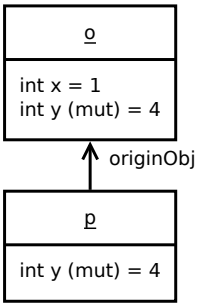


Fig. 3: Example of proxy object: p is a copy of o.

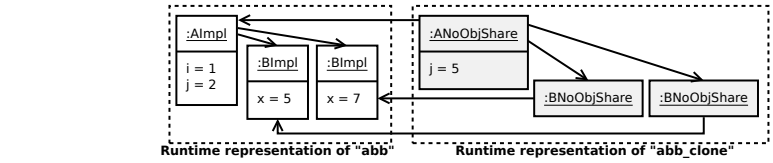
Algorithm 2: *cloning*

Data:
rr, a runtime representation of a model
strategy, the strategy used (*i.e.* what is shareable)
Result: rr_{clone}, a runtime representation of the clone

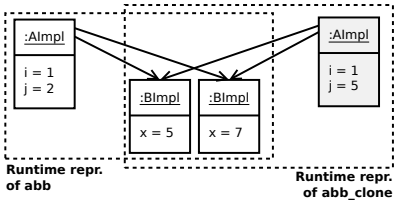
```
1  begin
2      rr_clone ← createEmptyRR()
3      copyMap← createEmptyMap()
4      for o ∈ rr do
5          if strategy.isObjShareable(o) then
6              rr_clone.addObject(o)
7              copyMap.addKeyValue(o, o)
8          else if strategy.isObjPartShareable(o) then
9              copy ← copyObjectProxy(o,strategy)
10             rr_clone.addObject(copy)
11             copyMap.addKeyValue(o, copy)
12         else
13             copy ← copyObject(o)
14             rr_clone.addObject(copy)
15             copyMap.addKeyValue(, o, copy)
16     resolveReferences(copyMap)
```

	Objects not shared (RQ #4 ok)	Objects shared (RQ #4 not ok)
Fields not shared (RQ #2 ok)	***DeepCloning***	***ShareObjOnly***
Fields shared (RQ #2 not ok)	***ShareFieldsOnly***	***ShareAll***

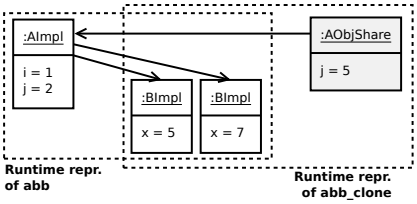
Table 1: Cloning operators obtained, one per strategy.



(a) *ShareFieldsOnly*



(b) *ShareObjOnly*



(c) *ShareAll*

Fig. 4: Runtime representations of models `abb` and `abb_clone` of Fig. 2 obtained with the different cloning operators.

a way to access to the shareable data. Fig. 3 illustrates this operation with a simple object o that has two fields x and y: x is not copied in p, but can still be accessed using the reference *originObj*.

The second operation is *cloning*(rr, *strategy*), the cloning algorithm itself, presented as Algorithm 2. It takes a runtime representation rr as input and a considered strategy, and returns a runtime representation rr_{clone} of a clone of the model of rr. Depending on the strategy outputs, each object is processed differently. If the object o is shareable, it is simply added in rr_{clone}, and is thus shared between rr and rr_{clone}. If o is partially shareable (not shareable but with shareable fields), a proxy copy of o is added to rr_{clone}. Finally, if o is not shareable at all, a regular copy is put in rr_{clone}.

4.3 Family of Cloning Operators

From our single cloning algorithm, we eventually obtain four cloning operators depending on the strategy used. We sum up the possibilities in Table 1, and we illustrate them with examples in Fig. 4. *DeepCloning* clones without any form of data sharing. *ShareFieldsOnly* clones using proxy objects to share as many fields as possible; Fig. 4a shows an example where each runtime object has a reference to the runtime object from which it originates. *ShareObjOnly* clones with object sharing only; Fig. 4b shows an example where `B` runtime objects are referenced by both models. Finally, *ShareAll* clones with both objects and fields sharing; Fig. 4c shows an example where only `j` is kept by the `A` runtime object.

In section 4.1, we listed four research questions to evaluate our cloning operators. Without proper benchmarking, we cannot answer the memory consumption (RQ #1) question yet. Concerning the efficiency when manipulating clones (RQ #2), we do not expect *ShareFieldsOnly* and *ShareAll* to comply because of

proxy objects. As they rely on of object sharing, *ShareObjOnly* and *ShareAll* are not compatible with generic operators that use the MOF `container()` reflective operation (RQ #4). However, our clones perfectly comply with the need to be manipulable by operations defined for the metamodel of the original model (RQ #3). This is illustrated by our implementation, which allows each clone to be manipulated using the EMF Java API generated for the metamodel.

4.4 EMF-Based Implementation

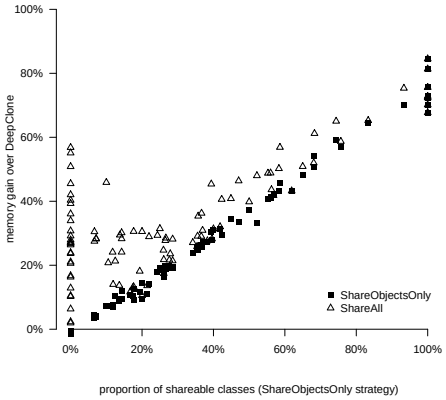
We implemented our approach in Java with as much EMF compatibility as possible, which required us to face two main challenges. First, we had to extend EMF libraries—including implementations of EObject and Resource—to ensure that *containment* references are handled consistently in each model. Second, our approach relies on proxy objects, which are easy to create dynamically using a prototype-based object oriented language. However, with a class-based object oriented language such as Java, the fields of an object are determined by its class at design-time. We thus have to generate appropriate classes beforehand, which we do with a java-to-java transformation using EMF and MoDisco [1] to remove non-shareable properties of generated EMF implementations. More details about the implementation can be found in the companion web page of the paper: `http://diverse.irisa.fr/software/modelcloning/`.

5 Evaluation and Results

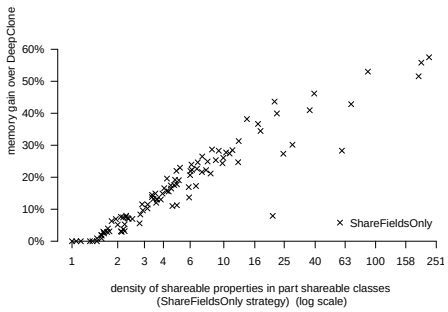
This section presents our evaluation. First we describe our dataset, then what we measure and the metrics considered for our metamodels, and finally the obtained results and how they relate to the requirements stated in Section 2.

5.1 Dataset

To evaluate this work, we need both various metamodels and models that conform to these metamodels. For the metamodels part, we developed a random Ecore model generator. We parameterized it the following way: a maximum number of 100 classes per metamodel, 250 properties per class and 50 mutable properties (which are properties with a `_m` suffix) per class. We use weighted randomness to create different kinds of properties, with the following weights: 30% of integers, 30% of booleans, 30% of strings, and 10% of references. For the models part, we generate for each metamodel a single model in a deterministic way that covers the whole metamodel. It starts from the roots, navigates through each composition and creates a maximum of two objects per encountered class. Then, all attributes are initialized with random values and references with random objects. We could have generated more models per metamodel, but our goal was to illustrate how our operators behave with varying *metamodels*, each with different shareable parts. For more information concerning the evaluation process you can refer to our companion web page `http://diverse.irisa.fr/software/modelcloning/`.



(a) Memory gain for the *ShareObjOnly* and *ShareAll* operators, with varying proportion of shareable classes



(b) Memory gain with *ShareFieldsOnly* against density of shareable properties in part. shareable classes (log scale)

Fig. 5: Memory gain results obtained for 1000 clones.

5.2 Measures

To verify that we reached our main objective, we must measure the memory consumption of the runtime representations of the clones, and more precisely the memory gain compared at the *DeepCloning* operator. For precise memory measures, we create a heap dump at the end of each evaluation run, and we analyze it using the Eclipse Memory Analyzer (MAT) [4]. The second measure we make is the read-access performance of the runtime representations of clones, compared to the one of the original model. We expect to see some performance decrease when proxy runtime objects are involved We proceed by measuring the amount of time required to navigate 10 000 times through each object of a model while accessing each of their properties.

5.3 Metrics

To embrace the variety of metamodels, we consider two metrics: the proportion of shareable classes when using either the *ShareObjOnly* or the *ShareAll* strategy, and the density of shareable properties within partially shareable classes when using the *ShareFieldsOnly* strategy. The first metric most likely correlates with the memory gain for operators that share objects, and the second for the operator that only shares fields.

5.4 Results

Each measure was done by creating the model of the metamodel, cloning it 1000 times with the chosen operator, and measuring both the memory footprint and the efficiency of one of the clones.

[4] `http://www.eclipse.org/mat/`

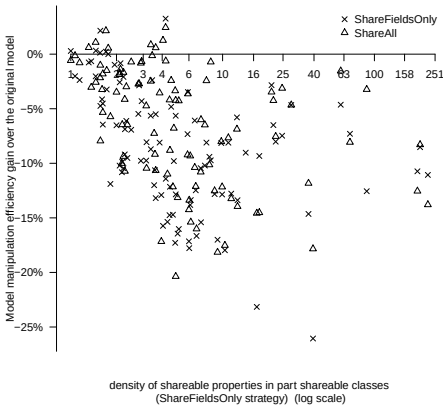


Fig. 6: Manipulation time gain for the *ShareFieldsOnly* and *ShareAll* operators, with varying density of shareable properties in part. shareable classes (log scale)

Fig. 5a shows the memory gain of the *ShareObjOnly* and *ShareAll* operators over the *DeepCloning* operator with varying proportion of shareable classes. We can see that the more shareable classes there are, the more memory gain there is. This relation appears linear for *ShareObjOnly*, and less regular for *ShareAll*. This is quite normal since the first operator only relies on object sharing, while the second is also influenced by the amount of shareable properties that can be shared through proxies. We also observe that *ShareAll* is always better that *ShareObjOnly*, which was expected since it shares fields in addition to objects. Some points may look surprising at position 0%, however they are simply caused by metamodels with very few classes and a high amount of shareable properties. Thus, sharing fields of such metamodels quickly gives very high gains.

Fig. 5b shows the memory gain of the *ShareFieldsOnly* operator over the *DeepCloning* operator with varying density of shareable properties within partially shareable classes. We observe a correlation between gain and the metric, and the gain raises up to approximately 40%. This operator gives overall worse results that the *ShareObjOnly* and *ShareAll* operators, but can give better results in some situations (e.g. metamodels with mostly partially shareable classes).

Finally, Fig. 6 presents the model manipulation efficiency gain over the runtime representation of the model originally cloned. We observe that, as expected because of the proxy design pattern, the operators *ShareFieldsOnly* and *ShareAll* both suffer from a little performance decrease. The median overhead is -9,5% for *ShareFieldsOnly* and -5.9% for *ShareAll*.

Overall, the results match our expectations. On the one hand, memory gain measures show that our operators are as good as *DeepCloning* when no parts are shareable, and are better and better as the quantity of shareable parts raises. Therefore, all our operators satisfy the need to reduce the memory footprint of clones (RQ #1). On the other hand, manipulation efficiency measures show that there is a little overhead when manipulating clones obtained by our operators *ShareFieldsOnly* and *ShareAll*. Thus, as we foresaw, these operators do not comply with the efficiency requirement (RQ #2).

5.5 Threats to Validity

We identified two main threats to our evaluation. First, using random metamodels, we hope to cover as many situations as possible in terms of metamodel design. Yet, have no way to be sure that our dataset contains enough "realistic" designs, as we have no metric for this criterion. Second, we use only one model per metamodel, which even if it covers the whole metamodel and is thus appropriate to evaluate our approach regarding metamodels characteristics, may overshadow some situations. For instance, if the objects of the model are mostly instances of non-shareable classes despite the fact that most classes are shareable, memory gain would not correlate with this metric as much as we observe.

6 Conclusion

Model cloning is an operation to duplicate an existing model that can be used in many kinds of applications. We identified four requirements for cloning operators: to be able to apply domain operators on clones, to have some memory gain over deep cloning, to be able to apply generic operators on clones, and to be able to manipulate clones as efficiently as their original model. Our goal was to provide cloning operators compliant with the first two requirements while satisfying the last two if possible. The approach we presented consists in sharing both runtime objects and fields between runtime representations of a model and its clones. We give four possible strategies to determine which parts of a metamodel are shareable, and we use these strategies to parameterize a generic cloning algorithm. We obtain four cloning operators, each being more appropriate for a specific situation. *DeepCloning* is the most basic operator with no memory footprint reduction, but that can be used in all situations where memory consumption is not an issue. *ShareFieldsOnly* shares fields of immutable attributes, which reduces the memory footprint of the clones but also introduces an overhead when manipulating them. *ShareObjOnly* shares objects to reduce significantly the memory footprint, but produced clones are not compatible with generic operations that rely on the `container()` specifiec in the MOF Reflection package. Finally, *ShareAll* shares both objects and remaining shareable fields, which saves even more memory, but with the weaknesses of the two previous operators. Our evaluation was done using a hundred randomly generated metamodels, and results show both memory gain over *DeepCloning* for all three other operators, and a loss of manipulation efficiency for *ShareObjOnly* and *ShareAll* operators.

To pursue this work, a possible direction would be to automate the choice of a cloning operator. For instance, it must be possible using static analysis of operations to determine whether the reflexive layer is used or not, and more precisely to detect the use of EMF `eContainer()`. This would give the possibility to automatically disable cloning operators that forbid the use of this operation.

Acknowledgement. This work is partially supported by the ANR INS Project GEMOC (ANR-12-INSE-0011).

References

1. Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 173–174, New York, NY, USA, 2010. ACM.
2. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1994.
3. Sherri Goings, Heather Goldsby, Betty H. C. Cheng, and Charles Ofria. An ecology-based evolutionary algorithm to evolve solutions to complex problems. In *Proc. of the Int. Conf. on the Simulation and Synthesis of Living Systems (ALIFE)*, 2012.
4. Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.
5. Gabor Karsai, Miklos Maroti, Akos Ledeczi, Jeff Gray, and Janos Sztipanovits. Composition and Cloning in Modeling and Meta-Modeling. *IEEE Transactions on Control Systems Technology*, 12(2):263–278, March 2004.
6. Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. Model transformation as an optimization problem. In *Model Driven Engineering Languages and Systems*, volume 5301 of *Lecture Notes in Computer Science*, pages 159–173. Springer Berlin Heidelberg, 2008.
7. Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPLSA '86, pages 214–223, New York, NY, USA, 1986. ACM.
8. Shahar Maoz. Model-based traces. In *Models in Software Engineering*, volume 5421 of *Lecture Notes in Computer Science*, pages 109–119. Springer Berlin Heidelberg, 2009.
9. OMG. Meta Object Facility (MOF) Core Specification, 2013.
10. Tripti Saxena and Gabor Karsai. MDE-Based Approach for Generalizing Design Space Exploration. In *Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, pages 46–60. Springer Berlin Heidelberg, 2010.
11. Dave Steinberg, Dave Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley, December 2008.
12. Robert Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972.